SPHERE: An Architecture for Secure Peer-to-Peer Hosted Encryption Record Exchange, Enabling Authenticatable, User-Controlled Decentralized Systems for Content Access

By Kenneth Lasyone

1. Abstract:

SPHERE is a decentralized system and architecture designed to provide identity authentication, content hosting, and permission validation without reliance on centralized infrastructure. It utilizes public storage of obfuscated private key fragments, reconstructable only through user-held credentials, to enable secure login and contact mutation. Static content and digital assets are distributed via a cryptographically verifiable hash table, with human-readable aliases resolved through a parallel DHT. Content blocks are signed by their creators, enforcing traceable authorship and immutability. Updates and sensitive operations require peer-issued, cryptographically signed tokens, enabling decentralized access control, behavior-based validation, and time-sensitive proofs. This trustless framework ensures censorship resistance, tamper-proof identity management, and incentivized peer participation across distributed networks.

---

2. Field of the Invention:

The present invention relates to decentralized identity authentication, peer-to-peer content distribution, and permissioned action validation in distributed networks. More specifically, it introduces a framework for secure login and identity management using obfuscated, publicly stored private key segments; serverless hosting of static digital content via distributed hash tables (DHTs); and a cryptographically verifiable token system that governs data mutation, submission rights, and computational task authorization without reliance on centralized control.

---

3. Description of the Related Art:

Conventional identity authentication systems rely heavily on centralized storage of credentials, exposing users to breaches, data leaks, and single points of failure. Blockchain-based approaches attempt decentralization but often depend on wallet-managed private keys or third-party authenticators, lacking usability, secure recovery options, and trustless validation mechanisms. These models fail to support segment-level identity reconstruction or encrypted multi-factor index structures within decentralized storage.

Similarly, traditional web hosting platforms—such as centralized servers and CDNs—introduce administrative bottlenecks, censorship risk, and critical reliance on third-party entities. While

decentralized storage systems like IPFS and Arweave offer partial solutions, they lack native support for authenticated content mutation, human-readable naming, or tight integration with decentralized identity frameworks. Protocols like DNS and HTTPS further reintroduce centralization through certificate authorities and domain registrars.

Moreover, current decentralized systems provide limited or ineffective permission control. Existing methods, such as staking models, fail to validate identity or enforce meaningful contribution. They do not offer peer-issued, cryptographically signed tokens for action gating, nor do they implement token scope, expiration, or inter-node accountability. As a result, these systems remain vulnerable to spam, abuse, and implicit reliance on centralized moderation mechanisms.

---

4. Summary of the Invention:

The present invention provides, without loss of generality, a system and method for decentralized identity authentication, content hosting, alias-based addressing, and cryptographically enforced permission control in peer-to-peer networks. It enables censorship-resistant, tamper-evident publishing and trustless user validation without centralized authorities. In one embodiment, the invention comprises:

- **A decentralized contact record structure**, stored in a distributed hash table (DHT), containing metadata, public keys, and encrypted private key segments—both real and decoy—paired with a navigable index decryptable only via user credentials.

- **An authentication mechanism** that uses user-provided credentials (e.g., username, password, PIN) processed through a key derivation function (KDF) to decrypt an index, enabling the reassembly of private keys embedded in the DHT.

- **A directional encryption scheme (BRACE)**—Byte-Routed Asymmetric Chain Encryption—that ensures key segments are securely linked and obfuscated, embedding routing data for ordered traversal and reconstruction.

- **An authentication validation subsystem** utilizing cryptographic hash and HMAC-based tuples, designed to confirm credential possession without transmitting plaintext secrets, and to render real validation tuples indistinguishable from decoys.

- **A dual-layer DHT architecture**, where a parallel Alias DHT maps human-readable names to cryptographic contact or content identifiers, enabling decentralized username and domain-style resolution.

- **A decentralized content storage system**, where static assets or web bundles are stored using deterministic content hashes, with updates signed and authorized by the

originating identity or authorized editors.

- **A token-gated permission framework**, where sensitive actions—such as content mutation, identity updates, or computational task execution—require cryptographically signed tokens issued by peer nodes.

- **A flexible token structure** embedding issuer and recipient IDs, expiration metadata, and usage scope, supporting fine-grained control over actions while enforcing mutual accountability between peers.

- **A verification protocol** requiring token presentation during sensitive operations, with signature validation and session-level authentication (e.g., via BRACE) to ensure origin integrity and action legitimacy.

- **A permission lifecycle model** supporting expiration, revocation, renewal, and time-delayed publishing windows, enabling rollback or cancellation of pending changes and enhancing network resilience.

- **A reputation-linked incentive system**, wherein nodes earn tokens by maintaining uptime, sharing blocks, or completing useful tasks, creating a merit-based gatekeeping layer that discourages abuse and enforces trustless access control.

The invention collectively enables a self-governing, decentralized ecosystem for identity management, secure login, serverless content hosting, and authenticated data mutation—achieving tamper resistance, censorship resilience, and permissioned operation without centralized oversight.

---

5. Brief Description of the Drawings

Figure 1: System Architecture Diagram
Illustrates the core components of the decentralized identity framework, including Contact Records, DHT-based storage, and peer-to-peer authentication pathways.

Figure 2: Login Decryption Flow
Depicts the login sequence using BRACE, showing how a user reconstructs their private key by decrypting obfuscated key segments based on entered credentials.

Figure 3: Contact Record DHT Structure
Outlines the structure of a Contact Record stored in the DHT, including metadata, public keys, encrypted key fragments, and authentication data used for secure identity validation.

Figure 4: Token-Based Publishing Validation
Shows the process of validating a publishing attempt using token-based permissions, including token verification and propagation of a signed content or identity block.

Figure 5: High-level content hosting system architecture:
Illustrates the high-level architecture of the decentralized web framework, including the interaction between the Content DHT, Alias DHT, and Contact DHT layers across participating nodes.

Figure 6: Content publishing and ContentID generation flow:
Shows the process of publishing static content to the DHT, including content hashing, ContentID generation, digital signature application, and storage within the network.

Figure 7: Alias resolution flow from alias to content retrieval:
Depicts the alias resolution workflow, demonstrating how a human-readable alias is converted into a hashed DHT key and used to retrieve the associated content block.

Figure 8: Content mutation flow using BRACE and token validation:
Outlines the content mutation sequence, including user authentication via BRACE, token presentation, and propagation of the updated content block with a new ContentID.

Figure 9: Redundancy and caching propagation between peer nodes:
Shows how content redundancy and caching propagate through the DHT as peer nodes retrieve, validate, and store replicated content blocks for fault tolerance and improved availability

---

6. Description Summary

6.1 Contact Record Structure

- MetaData: User-facing and system metadata fields, including display name, contact ID, version, and optional hashed identifiers

- Keys: Public-facing encryption and signature keys used for secure messaging and identity validation

- Key Fragments: Lists of encrypted private key segments, including true and decoy entries

- Index Structure: Encrypted navigation structure used to locate, decrypt, and reassemble valid key fragments

- AuthenticationData: Hash-based validation tuples used for account recovery and identity verification

- DHT Query and Resolution: XOR-based peer discovery process for locating and verifying contact blocks within the network

6.2 Alias Resolution and Contact Block Retrieval

- Alias (username) is entered by the user

- Alias is hashed to generate a 256-bit Kademlia-compatible ID

- Alias DHT is queried using this ID to retrieve the Alias Block

- Alias Block contains the corresponding ContactID

- ContactID is used to query the main Contact DHT

- Contact block is retrieved and verified for further use

6.3 Login Sequence

1. User submits username, password, and PIN
2. Credentials are used for decryption of the index
3. Index is used to traverse and decrypt true key segments
4. Segments are chained and reassembled
5. Resulting private key is matched against the Signature on the contact.
6. If valid, login is accepted; else, access is denied

6.4 Password Reset

- A failsafe mechanism allows the user to update their password if they still have the correct username, PIN, and access to the valid private key
- The system uses these to decrypt the original index and re-encrypt it using the new password

6.5 Contact Mutation Authorization (Optional Token-Based Control)

- Token requirement for contact creation or modification; validation by issuing node; prevents unauthorized writes.

6.6 Token-Gated Contact Editing

- Contact edits require the user to possess a token issued by the validating node
- The request and token are sent to the original node for validation
- Upon confirmation, the node pushes the update to the DHT

6.7 Content Record Structure

- **MetaData**: Human-readable and system-level metadata about the content, including the content title, contentID (analogous to a block ID), version tag, MIME type, and optional hashed descriptors (e.g., tags, creator ID, or content category).

- **Keys**: Public keys used to verify the content's authenticity or encrypt optional restricted components. This may include a signature key tied to the creator's Contact Record.

- **Key Fragments**: Encrypted private key segments used to authenticate edit access to the content. Includes both valid (true) and decoy (false) parts, obfuscated using BRACE logic.

- **Index Structure**: A securely encrypted structure containing traversal and reassembly instructions for the valid key fragments. Decryption of this index enables access control over content mutation.

- **AuthenticationData**: One or more cryptographic tuples (hash + HMAC) enabling verification of a user's authority to mutate the content, based on knowledge of partial credentials and/or the associated Contact private key.

- **Content:** The rendered or served body of the hosted object, typically consisting of static HTML, CSS, JavaScript, image, or media files. This field represents the current viewable version of the content that users interact with. Larger bundles may be compressed or referenced via manifest files to preserve DHT efficiency.

- **Content Hash**: A cryptographic digest (e.g., SHA-256) of the current content block or site bundle. This hash is used to ensure the integrity of the content during retrieval, replication, or audit. Any mutation to the content results in a new hash, facilitating tamper detection and versioning.

- **Content Signature:** A digital signature produced using the private key associated with the content creator's Contact Record. This signature is applied to the Content Hash and serves as a cryptographic guarantee that the content originated from a trusted, verifiable source. Signature validation enables readers or peer nodes to reject tampered or forged blocks.

6.8 Content Storage Model

- Each file, image, or static site is stored in a DHT-compatible content block
- Each block is hashed to produce a unique ContentID
- Large content can be split and referenced via manifest indexes
- Each block includes a digital signature and the creator's ContactID
- ContactID links the content to the creator's identity in the Contact DHT
- Signature verifies origin, integrity, and enables permission enforcement

6.9 Static Site Hosting

- Website components (HTML, CSS, JS, media files) are packaged and published into DHT content blocks
- Each site is assigned a ContentID based on a hash of the package or root file
- Human-readable aliases (e.g., ww4.sphere.site) map to ContentIDs via the Alias DHT
- Visitors query the Alias DHT, resolve to ContentID, and retrieve content from the DHT
- Each site includes a creator signature and ContactID for traceability
- Optional: Mutability features allow owners to update content using BRACE-authenticated login and token validation

6.10 Alias Mapping System

- A parallel distributed hash table (Alias DHT) maps human-readable domain-style aliases (e.g., **ww4.site.sphere—example only**) to static content hashes.
- The alias lookup mechanism mirrors the identity alias resolution model used in the authentication system.
- Optional version control may be implemented through timestamp metadata or chained content hashes to support mutable site history.

6.11 Mutation and Publishing Control

- Editing a hosted content block requires either possession of the associated private key or presentation of a cryptographically valid mutation token.
- Mutated blocks must be signed by the publisher's private key and are validated by the network before being propagated.

6.12 Tamper Resistance and Redundancy

- Hash-based addressing enforces immutability and prevents unauthorized alterations
- Redundant caching by peer nodes ensures content availability and fault tolerance
- All blocks undergo signature and integrity validation before acceptance or propagation

6.13 Decentralized Storage Viability

- Estimated volume of static public-facing content (excluding streaming/archives): ~120 PB
- Redundancy factor of 2x–3x yields target storage load of 240–360 PB
- With full user participation: storage per node falls below 1 GB
- Storage demands fall within modern device capabilities (phones, low-power PCs)
- Confirms feasibility of decentralized web hosting with high availability, fault tolerance, and global scalability

6.14 Identity-Scoped Subnet Overlays

- Nodes may join one or more isolated subnets defined by a unique bootstrap key and Contact namespace.
- Each subnet forms a cryptographically distinct trust graph with no global peer discovery required

- All authentication, alias resolution, and content access remain local to the subnet unless bridged
    Enables:
    - Private, organization-specific DHT clusters
    - Invisible mesh networks with scoped login and mutation rights
    - Multi-tenant or federated deployments without centralized oversight

- Subnet info may be embedded in ContactID prefixes, handshake payloads, or bootstrap keys

- Bridge nodes may exist with dual-authentication, enabling selective federation between overlays

6.15 Token Structure

- **Token ID:** Identifies the token itself is unique.

- **Issuer ID**: Identifies the node that issued the token; tied to the issuer's public signature key.

- **Recipient ID**: The node or identity authorized to use the token for specific actions.

- **Timestamp & Expiration**: Includes issuance time and an expiration window to limit token lifespan.

- **Signature**: A digital signature applied by the issuing node to validate authenticity and prevent tampering.

6.16 Token Issuance Logic

- Tokens are generated and signed only by nodes that have achieved valid bootstrap status.

- Tokens are issued after the recipient performs a validated network action (e.g., pings, DHT retrievals, uptime).

- Issuers are prohibited from generating tokens for themselves.

6.17 PingPal: a system for uptime token issuance.

- The Node chooses a peer as a **PingPal**
- The node pings their **PingPal** a **PingPal** request
- The **PingPal** then store the request from the peer
- The peer in 24 hours can re ping their **PingPal** and get rewarded a token

6.18 Token Validation and Enforcement

- Tokens must be presented alongside sensitive requests (PUT, EDIT, COMPUTE).
- Each token is verified using the issuer's public signature key.
- Validation includes expiration check, scope compliance, and recipient match.
- Requests lacking valid tokens are rejected.

6.19 Token Lifecycle Management

- Tokens may be **extended** via push-token ping-pong handshakes between nodes.
- Expired tokens are moved into a pending removal list and eventually purged.
- Revoked or reused tokens are tracked and rejected to prevent replay attacks.

6.20 PushToken Extend Handshake

- Issuing nodes send a **PushTokenExtendPing** to the recipient.
- Recipients respond with the original token, proving possession and validity.
- Upon verification, token expiration is extended.
- Failure to respond results in the token being scheduled for removal.

6.21 Token Use Cases

- **Content Mutation**: Required for publishing or editing DHT-hosted content blocks.
- **Identity Mutation**: Required for updating Contact Records in the identity system.
- **Task Submission**: May be required to submit work to computation clusters or participate in consensus.
- **Alias Registration**: May gate human-readable alias creation or updates.

6.22 Reputation-Based Earning Model

- Nodes earn tokens by performing useful tasks for the network.
- Actions are verified before token issuance.
- Issuance conditions can include ping success, data replication, or uptime benchmarks.
- Token distribution is trustless and peer-driven, without central coordination.

6.23 Abuse Prevention and Token Safeguards

- Self-issued tokens are cryptographically invalidated.
- Nodes track issued token histories and enforce non-reissuance.
- Scope and expiration prevent overreach or misuse.
- Replay protections guard against token reuse across contexts.

---

6. Detailed Description

6.1. Contact Record Structure

Each user within this implementation of the Peer- to- Peer Kademlia network are represented by a uniquely addressable **Contact Record**, stored in a decentralized hash table (DHT). This record serves as the cryptographic anchor for identity, authentication, communication, and verification processes. The contact data is **publicly queryable**, but all sensitive content is encrypted and obfuscated to prevent unauthorized access or inference.

6.1.1 MetaData

The **ContactMetaData** object contains descriptive and optional fields to identify or provide context for the contact. This includes, but is not limited to:

- **DisplayName:** A user-facing name or alias.
- **Name:** The user's actual or system-registered name.
- **ContactID:** The unique identifier of the contact (also referred to as the "Block ID").
- **ContactVersion:** A version tag that allows future-proofing and backward compatibility as the contact format evolves.
- Optional hashed personal identifiers such as:
    - **HashedSSN**
    - **HashedCardNumber**
    - **HashedAccountNumber**
    - **HashedRoutingNumber**
      These fields may be used by third-party applications to verify sensitive user information without ever exposing the raw values. Hashes are salted and secured, enabling one-way validation for things like secure payments, banking connections, or KYC.

Additional optional fields may include language preferences, contact information (email, phone), avatar reference hash, and a brief personal description.

6.1.2 Keys

The **ContactKeys** structure holds the asymmetric and symmetric keys used for message encryption, personal signature validation, and local contact protection. These keys are all public-facing but play unique roles:

- **SemiPublicKey:** A semi-public identity key used for shared access and contact decryption authorization.

- **EncryptedLocalSymmetricKey:** A symmetric key encrypted using trusted public infrastructure and used to decrypt the contact locally.

- **PublicPersonalEncryptionKey:** Used for encrypting personal messages or data sent to the contact. Only the owner's private decryption key can access this content.

- **PublicPersonalSignatureKey:** Used by other peers to validate any data or message the contact signs using their private signature key.

6.1.3 Key Parts

Each contact record stores two lists of encrypted private key fragments:

- **KeySignatureParts**
    - **Alpha**- A three byte array used to identify the starting point for selection and decryption.
    - **KeyParts**- List of all the encrypted parts(True and Decoy).
    - **SplitIndex**- Used to track the order the keys need to be placed in for reassembly.

- **KeyEncryptionParts**
    - **Alpha**- A three byte array used to identify the starting point for selection and decryption.
    - **KeyParts**- A list of all the encrypted parts(True and Decoy).
    - **SplitIndex**- Used to track the order the keys need to be placed in for reassembly.

Alpha is the **Routing Bytes** of the first true segment, the starting point in the list of **KeyParts** for decryption and reassembly.

KeyParts is an array containing a mix of **True Key segments** (which together reconstruct the user's private keys) and **Decoy Key segments** (noise, false data). The purpose is to **obfuscate the position and identity of real segments**, making brute-force or inference-based attacks computationally infeasible.

SplitIndex is a number that represents the order the true key order was shuffled to after being split into the three parts.

6.1.4 BRACE (Byte-Routed Asymmetric Chain Encryption) Protocol

BRACE** (Byte-Routed Asymmetric Chain Encryption) **Protocol is a method of encrypting dynamically split key segments in a directional chain, where each segment embeds routing metadata (in the form of leading bytes) to identify the subsequent segment in the decryption process. This obfuscates the sequence of real key fragments among decoys while enforcing strict reconstruction logic through embedded byte-based identifiers. BRACE ensures that only the user possessing the correct credentials can follow the decryption path to reassemble the original private key.

KeySplitting-The full private key (FPK) is taken along with the user's password and pin at the time of contact creation. The FPK is then split into 3 dynamically sized segments. These three key segments are then shuffled into **A**, **B**, and **C**, to ensure random processing for encryption. The SplitIndex is stored for key part reassembly in the correct order upon logging on.

The shuffled key segments are then processed via chained directional encryption with embedded byte routing.

Byte-Routed Asymmetric Chain Encryption

1) Key segment **C** is encrypted using part **B** as the key and the Hash of part **A** as the salt.
2) The first 3 bytes (routing bytes) of the encrypted **C** are added to the front of **B**.
3) **B** is encrypted using **A** as a key and a hash of the user password as the salt,
4) The first 3 bytes (routing bytes) of encrypted **B** are added to the front of **A**
5) **A** is then encrypted with the user's password using the user's pin as the salt.
6) The first 3 bytes (routing bytes) of **A** is stored as **Alpha** on the **EncryptedKeyBytes**

Routing Bytes- Routing bytes are used during reconstruction to identify the next segment in the Decryption and reassembly process.

This method ensures that **only the rightful user**—armed with the proper credentials—can decrypt the correct path through the segment list, bypassing decoys and reconstructing the usable private key.

6.1.5 AuthenticationData

The **AuthenticationData** field contains hash and HMAC-validation authentication tuples used to verify a user's knowledge of the correct credentials without revealing or storing the credentials themselves.Each tuple is derived by hashing a user credential (such as a password or PIN) along with the full private key and contactID, as well as a corresponding HMAC generated using the private key as the secret. These tuples are used during login or password reset processes to validate the user's knowledge of one or more secrets.

The system stores both valid and decoy (false) tuples in the **AuthenticationData** structure, each formatted identically, to prevent pattern detection or inference by external observers. Only users who possess the correct private key and at least one valid credential (password or PIN) can regenerate a hash that, when passed through the HMAC function, matches one of the stored validation entries.

This ensures that authentication is:

- **Decentralized**: no central authority validates credentials
- **Resilient**: users can validate themselves using partial credentials and cryptographic proof
- **Obfuscated**: real authentication entries are hidden among randomized decoys, making it computationally infeasible to identify valid hashes without legitimate inputs

6.1.6 DHT Query and Resolution

When a node requests a contact:

1. It first checks if the requested contact block is available locally in its cache. If found, it proceeds directly to validation and usage.
2. If unavailable, it initiates a **GetRequest** across the network using XOR-based proximity routing as defined by the Kademlia protocol. The request uses the hashed ContactID as the lookup key, ensuring deterministic routing toward the closest-known nodes.
3. The requesting node receives a set of candidate contact blocks from multiple peer nodes. Each block is:
    - Verified for authenticity using embedded digital signatures or integrity hashes
    - Compared against redundancy or quorum thresholds if multiple versions are received
    - Stored locally for future queries and reduced latency
4. Once the correct contact block is identified and validated, the system proceeds to use the embedded BRACE-encrypted key parts for authentication and identity verification.

6.2 Alias Resolution and Contact Block Retrieval

To provide a human-friendly mechanism for addressing users within the SPHERE network, each user may register one or more **Aliases**, also referred to as usernames or handles. These

aliases are resolved through a separate, parallel **Alias DHT** that maps readable usernames to the underlying unique contact block ID (ContactID).

6.2.1 Alias Block Construction

Each alias is stored as a separate DHT block containing:

- The alias string (optionally hashed again for validation)
- The associated ContactID
- A digital signature of the alias block contents
- The public key corresponding to the signer
- Optional metadata such as a timestamp, version identifier, or expiration marker

Alias blocks are addressed via a deterministic identifier generated by hashing the alias string using a cryptographic hash function (e.g., SHA-256) to produce a 256-bit Kademlia-compatible key. This ensures consistent, collision-resistant routing and lookup behavior within the decentralized network.

6.2.2 Lookup Process

This alias resolution mechanism operates on a Kademlia-based DHT, which enables fast, logarithmic-time lookups with O(log N) complexity. As the network grows, lookup performance remains efficient due to Kademlia's XOR-based proximity routing and recursive peer discovery logic.

When a user enters an alias to initiate a login, message, or profile request:

1. The alias string is normalized and hashed into a 256-bit key.
2. The querying node sends a **GetRequest** for that key to the Alias DHT.
3. If the alias exists, the network responds with the corresponding Alias Block.
4. From the Alias Block, the system retrieves the user's **ContactID**.
5. The system then queries the **main Contact DHT** for the full contact record using the provided ID.
6. If the contact record is found and verified, it is loaded for decryption or interaction.

This mechanism decouples user-facing identity from internal system addressing, enabling:

- Easy changes of alias without altering the underlying contact data
- Fast, readable lookups
- Lightweight contact sharing (e.g., "find me at Kenny#147")

6.2.3 Decentralization and Tamper Resistance

Alias blocks are publicly stored within the decentralized hash table (DHT), allowing for distributed access without reliance on a central authority. Each alias block is cryptographically validated upon retrieval, ensuring that tampering or unauthorized modification can be detected immediately.

When implemented, digital signatures and signer metadata allow nodes to authenticate the origin of an alias block, verifying that it was created or last modified by the rightful key owner.

This architecture provides the following guarantees:

- **Trustless resolution**: Alias lookups can be performed without requiring trust in any specific node or intermediary
- **Censorship resistance**: Alias blocks cannot be selectively blocked or overwritten without violating DHT consensus rules or signature validation
- **Ownership protection**: Aliases cannot be hijacked or reassigned without possession of the original private key
- **Collision handling**: Hash-based addressing ensures that conflicting aliases deterministically resolve to unique network keys, preventing ambiguous lookups

6.3 Login Sequence

The login process follows a multi-factor, cryptographically validated sequence designed for complete decentralization and tamper resistance. In this embodiment, the following steps occur:

1. **Credential Submission**
   The user provides a username (alias), password, and PIN.

2. **Alias Resolution**
   The alias is normalized and hashed into a DHT-compatible key. A lookup is performed in

the Alias DHT to retrieve the corresponding ContactID. This ID is then used to query the Contact DHT to retrieve the user's contact record.

3. **EncryptedKeyParts and Alpha Retrieval**
   The **EncryptedKeyParts** are pulled from the **Contact** and decrypted using the user's password as the key and using the pin as the salt. The decrypted **EncryptedKeyParts** reveals **Alpha** and a **SplitIndex**.

4. **Key Segment Traversal via BRACE**
   The system identifies and decrypts the true key segments by following embedded routing bytes, starting at the known **Alpha** and **BRACE (Byte-Routed Asymmetric Chain Encryption)** logic. Each segment is decrypted in reverse order using the proper salts and key material.

5. **Private Key Reconstruction**
   The decrypted segments are reassembled in the correct order using the SplitIndex. The resulting byte sequence is the full private key.

6. **Validation**
   The reconstructed private key is verified by re-signing the hashed Contact Record and comparing the resulting signature to the one stored in the record. If the signatures match, the key is deemed valid and authentication is successful.

6.4 Password Reset

Upon creation of the contact the user is provided an option to export their private key. This key is then stored securely by the user. If the user forgets their password or pin, the user can provide the remaining known element (password or pin) and their copy of the private key to reset the password and pin.

During password reset, the system leverages the **AuthenticationData** field to validate user authenticity prior to decrypting the BRACE-encrypted key segments. A successful HMAC match confirms knowledge of at least one valid credential and enables the system to proceed with index decryption and private key reconstruction. In password or PIN recovery scenarios,

AuthenticationData provides a secure verification step without requiring centralized recovery mechanisms or trusted intermediaries, thereby maintaining the decentralized integrity of the overall authentication framework.

6.5 Contact Mutation Authorization (Optional Token-Based Control)

Write operations on contact records—such as the creation, modification, or deletion of identity metadata or key fragments—may require a cryptographically signed token. This token may be issued by a peer node within the network and used to authorize the mutation before the updated contact block is published to the decentralized hash table.

The specifics of token issuance, structure, and validation are described in a separate system for decentralized token-based governance and permission control. This mechanism may be used to prevent unauthorized updates, mitigate spam, or enforce trust-based access policies within the network.

6.6 Token-Gated Contact Editing

A user seeking to modify their contact record must first complete a successful login using the BRACE protocol. Once authenticated, the system allows access to a contact editing interface.

To publish the mutation, the user must present a valid, cryptographically signed token. This token is verified by the node responsible for validating and propagating the update. Upon successful verification of the token and the integrity of the modified contact block, the node publishes the updated block to the decentralized hash table.

The mutation process may include a delay period (e.g., 24 hours) to allow for cancellation or rollback in the event of unauthorized access or user error. The underlying token issuance, expiration, and validation logic are governed by a separate decentralized permission control framework.

The system is not limited to any specific cryptographic algorithm, encryption scheme, data segmentation model, or DHT routing protocol. Alternate implementations may employ different key derivation functions, hash algorithms, or network architectures, provided they preserve the fundamental properties of verifiable identity, decentralized storage, and tamper-evident access control.

6.7 Content Record Structure

- **MetaData**: Human-readable and system-level metadata about the content, including the content title, contentID (analogous to a block ID), version tag, MIME type, and optional hashed descriptors (e.g., tags, creator ID, or content category).

- **Keys**: Public keys used to verify the content's authenticity or encrypt optional restricted components. This may include a signature key tied to the creator's Contact Record.

- **Key Fragments**: Encrypted private key segments used to authenticate edit access to the content. Includes both valid (true) and decoy (false) parts, obfuscated using BRACE logic.

- **Index Structure**: A securely encrypted structure containing traversal and reassembly instructions for the valid key fragments. Decryption of this index enables access control over content mutation.

- **AuthenticationData**: One or more cryptographic tuples (hash + HMAC) enabling verification of a user's authority to mutate the content, based on knowledge of partial credentials and/or the associated Contact private key.

- **Content:** The rendered or served body of the hosted object, typically consisting of static HTML, CSS, JavaScript, image, or media files. This field represents the current viewable version of the content that users interact with. Larger bundles may be compressed or referenced via manifest files to preserve DHT efficiency.

- **Content Hash**: A cryptographic digest (e.g., SHA-256) of the current content block or site bundle. This hash is used to ensure the integrity of the content during retrieval, replication, or audit. Any mutation to the content results in a new hash, facilitating tamper detection and versioning.

- **Content Signature:** A digital signature produced using the private key associated with the content creator's Contact Record. This signature is applied to the Content Hash and serves as a cryptographic guarantee that the content originated from a trusted, verifiable source. Signature validation enables readers or peer nodes to reject tampered or forged blocks.

6.8 Content Storage Model

- Each file, image, or static site is stored in a DHT-compatible content block
- Each block is hashed to produce a unique ContentID
- Large content can be split and referenced via manifest indexes
- Each block includes a digital signature and the creator's ContactID
- ContactID links the content to the creator's identity in the Contact DHT
- Signature verifies origin, integrity, and enables permission enforcement

6.9 Static Site Hosting

- Website components (HTML, CSS, JS, media files) are packaged and published into DHT content blocks
- Each site is assigned a ContentID based on a hash of the package or root file
- Human-readable aliases (e.g., ww4.sphere.site) map to ContentIDs via the Alias DHT
- Visitors query the Alias DHT, resolve to ContentID, and retrieve content from the DHT
- Each site includes a creator signature and ContactID for traceability
- Optional: Mutability features allow owners to update content using BRACE-authenticated login and token validation

6.10 Alias Mapping System

- A parallel distributed hash table (Alias DHT) maps human-readable domain-style aliases (e.g., **ww4.site.sphere—example only**) to static content hashes.
- The alias lookup mechanism mirrors the identity alias resolution model used in the authentication system.

- Optional version control may be implemented through timestamp metadata or chained content hashes to support mutable site history.

6.11 Mutation and Publishing Control

- Editing a hosted content block requires either possession of the associated private key or presentation of a cryptographically valid mutation token.

- Mutated blocks must be signed by the publisher's private key and are validated by the network before being propagated.

6.12 Tamper Resistance and Redundancy

- Hash-based addressing enforces immutability and prevents unauthorized alterations

- Redundant caching by peer nodes ensures content availability and fault tolerance

- All blocks undergo signature and integrity validation before acceptance or propagation

The system is intended to support a wide range of cryptographic primitives, storage topologies, and content structures, including but not limited to implementations yet to be developed, so long as they preserve the principles of decentralized integrity, verifiable authorship, and censorship resistance.

6.7 Content Record Structure

Each hosted asset (static site, file, or bundle) within this implementation is represented by a uniquely addressable Content Record, stored in a decentralized hash table (DHT). The Content Record serves as the cryptographic anchor for hosting, editing, signature validation, and retrieval tracking. The block may optionally include mutation constraints such as identity verification, routing logic, and token-based permissions.

6.7.1 MetaData

The **ContentMetaData** object provides descriptive and system-relevant fields to identify, classify, and manage the hosted data. These fields may include:

- **ContentTitle:** A user-facing title or label for display purposes.

- **ContentID:** A unique identifier, derived from the content's cryptographic hash, used to locate and address the block.

- **ContentVersion:** A semantic or numeric tag allowing for version control or rollback.

- **MIMEType:** A type declaration (e.g., text/html, image/png) indicating how the content should be interpreted.

- **Optional Descriptors:** Hashed values such as creator ID, tags, or classifications that assist in search and filtering without revealing direct values.

6.7.2 Keys

The ContentKeys section contains public-facing keys that may be used for:

- Verifying digital signatures applied to the content block.

- Encrypting restricted parts of the content (e.g., gated components or private metadata).

- Associating the content with the public key of the creator's Contact Record, enabling traceability and trust validation.

6.7.3 Key Parts

Each contact record stores two lists of encrypted private key fragments:

- **KeySignatureParts**
    - **Alpha**- A three byte array used to identify the starting point for selection and decryption.
    - **KeyParts**- List of all the encrypted parts(True and Decoy).
    - **SplitIndex**- Used to track the order the keys need to be placed in for reassembly.

- **KeyEncryptionParts**
    - **Alpha**- A three byte array used to identify the starting point for selection and decryption.
    - **KeyParts**- A list of all the encrypted parts(True and Decoy).
    - **SplitIndex**- Used to track the order the keys need to be placed in for reassembly.

Alpha is the **Routing Bytes** of the first true segment, the starting point in the list of **KeyParts** for decryption and reassembly.

KeyParts is an array containing a mix of **True Key segments** (which together reconstruct the user's private keys) and **Decoy Key segments** (noise, false data). The purpose is to **obfuscate the position and identity of real segments**, making brute-force or inference-based attacks computationally infeasible.

SplitIndex is a number that represents the order the true key order was shuffled to after being split into the three parts.

6.7.4 AuthenticationData

The **AuthenticationData** field contains hash and HMAC-validation authentication tuples used to verify a user's knowledge of the correct credentials without revealing or storing the credentials themselves.Each tuple is derived by hashing a user credential (such as a password or PIN) along with the full private key and contactID, as well as a corresponding HMAC generated using the private key as the secret. These tuples are used during login or password reset processes to validate the user's knowledge of one or more secrets.

The system stores both valid and decoy (false) tuples in the **AuthenticationData** structure, each formatted identically, to prevent pattern detection or inference by external observers. Only users who possess the correct private key and at least one valid credential (password or PIN) can regenerate a hash that, when passed through the HMAC function, matches one of the stored validation entries.

This ensures that authentication is:

- **Decentralized**: no central authority validates credentials
- **Resilient**: users can validate themselves using partial credentials and cryptographic proof
- **Obfuscated**: real authentication entries are hidden among randomized decoys, making it computationally infeasible to identify valid hashes without legitimate inputs

6.7.5 Content

The **Content** field represents the core payload served to end users, typically comprising static web assets such as HTML, CSS, JavaScript, media files, documentation, or other downloadable resources. This content may include pre-rendered site pages or interactive elements designed to function independently of server-side logic. To support efficiency within the decentralized network, content may be optionally compressed, chunked, or linked through manifest structures for scalable delivery. The data stored in this field constitutes the live-facing experience users receive upon accessing the associated ContentID or alias.

This is the core data served to users. It may include:

- HTML/CSS/JS site code, media files, documentation, or downloadable content.
- Pre-rendered outputs or user-facing interfaces bundled in static form.
- Optionally compressed, chunked, or manifest-linked for performance and DHT efficiency.

Content blocks may reference one another via embedded URIs or content hashes, enabling modular application structures, decentralized dependencies, or hyperlinked document graphs.

Content may optionally link to an identity system, including but not limited to decentralized identifiers (DIDs), zero-knowledge credentials, or third-party verification protocols.

6.7.6 Content Hash

The Content Hash is a cryptographic digest, typically generated using SHA-256 or a stronger algorithm, applied to the entire content body or the aggregated manifest in cases of multi-block content. This hash serves as a unique, tamper-evident identifier for the content block, ensuring integrity during retrieval, replication, or network-wide audit processes. Because the hash is deterministic, any modification to the underlying content results in a new hash, thereby preventing unauthorized alterations and enforcing immutability across the decentralized storage system.

6.7.7 Content Signature

The **Content Signature** is a digital signature created using the content creator's private key at the time of publishing. This signature is applied to the Content Hash, and optionally to associated metadata fields, to provide cryptographic proof of authorship and ensure that the content originated from a verified source. By signing both the content and selected metadata, the system enables recipients or peer nodes to validate the authenticity of the block and reject forged or tampered data, preserving the trust and integrity of the decentralized publishing model.

6.8 Content Storage Model

All static content—such as files, images, or entire websites—are encapsulated within DHT-compatible blocks for decentralized distribution and retrieval. Each block is individually hashed using a cryptographic algorithm (e.g., SHA-256) to generate a deterministic, tamper-evident identifier known as the **ContentID**. This ID functions as the addressable locator for the block within the decentralized network and ensures that any alteration to the content results in a new, distinct identifier.

For larger sites or applications consisting of multiple resources, content may be segmented across multiple blocks and organized through a **manifest index**. This index acts as a map that maintains references to all component blocks, preserving their order and enabling seamless reconstruction by clients.

Every block includes a **digital signature** generated by the content creator's private key at the time of publishing. This signature is applied to the **Content Hash** and optionally to block-level metadata, providing cryptographic proof of authorship and preventing unauthorized tampering. The block also includes the creator's **ContactID**, which links back to the corresponding **Contact Record** in the identity system. This association facilitates trust establishment, origin tracing, and enforcement of permission policies such as content mutation or ownership validation.

By integrating these elements—hash-based addressing, manifest indexing, and signature verification—this model provides a robust, decentralized alternative to traditional content hosting while preserving provenance, auditability, and security.

While described herein using a Kademlia-compatible DHT, the storage layer may also leverage alternative decentralized systems, such as gossip-based replication, DAGs, or consensus-backed ledgers, where appropriate.

6.9 Static Site Hosting

The system is not limited to static content and may be extended to support dynamic applications, user-generated content streams, or computer-generated outputs, provided the content can be deterministically hashed or versioned.

In this embodiment, websites are composed of static components—such as HTML, CSS, JavaScript, and media files—which are bundled and published into DHT-compatible content blocks. These blocks are individually addressable and retrievable from the network using a cryptographic hash derived from the content itself. The resulting **ContentID** serves as the unique reference to the root or manifest block of the site.

To enable user-friendly access, the system incorporates an **Alias Mapping System** that links human-readable addresses (e.g., **ww4.sphere.site**) to the associated ContentID. This mapping is maintained in a parallel **Alias DHT**, allowing users to resolve aliases to deterministic block identifiers without relying on centralized DNS infrastructure. Once the alias is resolved, the user's node retrieves the appropriate content block(s) from the main DHT and renders the site locally.

Each published site includes the **ContactID** of the creator and a **digital signature** over the site's Content Hash. This ensures content authenticity, traceability, and allows third-party verification of the publisher's identity through the decentralized Contact system. By anchoring the site to a known Contact Record, the system enables trust and accountability without sacrificing decentralization.

For site owners requiring update capabilities, **mutability** is optionally supported through the same mechanisms described in the Contact system. This includes BRACE-encrypted authentication workflows and token-gated mutation rights, ensuring that only authorized users can alter previously published content. Update attempts are verified cryptographically and may include a delay period for rollback, mirroring the protections established in identity-based editing workflows.

6.10 Alias Mapping System

To provide human-readable access to decentralized content, the system employs a parallel **Alias DHT** that maps easily recognizable aliases (e.g., ww4.site.sphere—example only) to the corresponding **ContentID** hashes of hosted static sites. This decouples the user-facing naming convention from the underlying cryptographic identifiers used within the DHT, enabling intuitive navigation and simplified sharing.

The alias resolution model is derived from the system's identity framework, wherein each alias is hashed deterministically (e.g., using SHA-256) to form a **DHT-compatible key**. This key is used to locate the corresponding **Alias Block** containing the mapped ContentID, optional metadata, and an optional digital signature for validation.

Alias blocks may also include **versioning information**, such as timestamped references or **chained content hashes**, to support mutable content histories. This allows users to reference or retrieve past versions of a site, while maintaining a consistent and trustless mechanism for resolving the most current version.

By separating human-readable identifiers from the storage model, and cryptographically binding each alias to content via signed mappings, the system supports censorship resistance, ownership verification, and mutable site evolution—all without reliance on centralized DNS or certificate authorities.

6.11 Mutation and Publishing Control

To preserve content integrity while enabling authorized updates, the system incorporates a mutation control framework based on identity verification and token validation.

Content mutation—such as editing an existing site, updating a file, or replacing a manifest—requires the publisher to demonstrate ownership of the original content. This is achieved by requiring the user to authenticate through the **BRACE login process**, reassemble the associated private key, and match it to the public key embedded in the original content's signature.

In addition to verifying identity, the system requires a **cryptographically signed mutation token**. This token is issued by a trusted node and encodes permissions, expiration time, and potentially a usage scope (e.g., content type or domain segment). The mutation token ensures that only users with validated authorization can perform write operations within the DHT.

Once authenticated, the modified content is re-hashed and re-signed using the user's private key. A **new ContentID** is generated, and the updated block is propagated to the network. The corresponding alias may be updated to point to the new version, or versioning logic (e.g., chained hashes) can be employed to maintain access to older revisions.

To enhance safety and support dispute mitigation, mutations may undergo a **publishing delay period** (e.g., 24 hours), during which they can be canceled or challenged by the original author. This mechanism provides a safeguard against stolen keys or rogue mutation attempts, without relying on centralized moderation.

Through these combined controls—authenticated access, token gating, and cryptographic proof—the system ensures that publishing rights are restricted to verified owners, while preserving the decentralized nature and trustless mutability of the content network.

Other permission mechanisms, such as cryptographic challenge-response schemes or stake-based authorizations, may be used in place of or in addition to token validation.

6.12 Tamper Resistance and Redundancy

The system ensures the long-term integrity, availability, and authenticity of hosted content through a combination of cryptographic guarantees and decentralized replication strategies. Redundancy strategies may include passive caching, active mirroring, erasure coding, or incentive-driven replication layers.

Each content block is identified and addressed by a **cryptographic hash** of its contents (ContentID), typically using SHA-256 or a stronger algorithm. This guarantees **immutability**, as any change to the content will result in a different hash, making it impossible to overwrite or disguise tampered data without detection. Peers that retrieve content validate its hash upon receipt, rejecting any block that fails verification.

To provide **tamper resistance**, every block also includes a **digital signature** generated using the private key of the creator. This signature is applied to the Content Hash (and optionally metadata), enabling peers to verify both the origin and the legitimacy of the content. If the signature does not match the ContactID referenced within the block, the content is deemed invalid.

Content is replicated across the decentralized hash table using **Kademlia's proximity-based routing and passive caching**. When a node requests or relays a content block, it stores a copy

locally. Over time, this leads to **redundant caching** among geographically and topologically diverse peers, significantly increasing availability and fault tolerance. Even if the original publisher goes offline, other nodes can continue to serve the content.

Together, these mechanisms ensure that:

- Content cannot be silently modified without detection.
- Identity-linked signatures bind data to verified authorship.
- Redundant peer storage protects against data loss or takedowns.

This design upholds the decentralized principles of the network while providing robust protection against forgery, censorship, and data degradation.

6.13 Decentralized Storage Viability

The total volume of static public-facing content on the internet—excluding streaming media and deep archival data—is estimated to be approximately 120 petabytes (PB). To accommodate for fault tolerance, multi-region availability, and eventual consistency, a redundancy factor of 2x to 3x is employed, yielding an effective decentralized storage target between 240 and 360 PB.

As of 2025, global internet adoption exceeds 5 billion users. Distributing the entire content load, even among a subset of participants, demonstrates technical feasibility. For illustrative purposes:

- If 1% of users (i.e., 50 million nodes) participate in content replication, the per-node burden equates to:
    - ~4.8 GB at 2x redundancy
    - ~7.2 GB at 3x redundancy

Modern consumer devices, including mobile phones and low-power computers, routinely exceed this capacity, confirming viability under limited participation.

However, under a fully decentralized model, all participants share responsibility for content distribution. In embodiments where all users contribute, storage requirements per node fall to under 1 GB, even under high redundancy factors.

This model affirms the practicality of a decentralized web in which data integrity, availability, and hosting accountability are distributed systematically across the user base, rather than concentrated within centralized servers or service providers.

The system is not limited to any specific cryptographic algorithm, encryption scheme, data segmentation model, or DHT routing protocol. Alternate implementations may employ different key derivation functions, hash algorithms, or network architectures, provided they preserve the fundamental properties of verifiable identity, decentralized storage, and tamper-evident access control.

6.14 Identity-Scoped Subnet Overlays

Nodes may participate in one or more identity-scoped subnet overlays. Each overlay is defined by a unique bootstrap key and Contact namespace, allowing nodes to form cryptographically isolated trust environments within the broader protocol. These subnets operate independently, with no requirement for global visibility or peer discovery across overlays. All authentication, alias resolution, and content addressing are confined to the subnet unless explicitly bridged by a cross-signed peer.

This enables:

- Private, organization-specific DHT clusters
- Invisible mesh networks with scoped login and mutation rights
- Multi-tenant or federated deployments without centralized control

Overlay identifiers and routing filters may be embedded in the ContactID prefix, initial handshake payload, or included in the subnet bootstrap key itself. Nodes may optionally serve as bridges between overlays when dual-authenticated, allowing federation without full merging of trust graphs.

6.15 Token Structure

Each action-regulating token is represented by a uniquely signed, cryptographically verifiable structure. Tokens are issued by peer nodes and used to gate sensitive operations such as publishing, editing, or task execution. Tokens are designed to be **lightweight, traceable, and tamper-resistant**, and they function as the decentralized enforcement mechanism for permission control across the network. Each token includes the following components:

6.15.1 TokenID

The TokenID is a 256-bit globally unique identifier assigned to each token at the time of issuance. This ID may be randomly generated, derived from a cryptographic nonce, or deterministically calculated using hash functions.

The TokenID allows nodes to track token usage and enforce non-reusability, enabling replay protection and duplication detection across the network.

6.15.2 IssuerID

The **IssuerID** identifies the node that created and signed the token. It provides the necessary anchor for token verification through peer published mutations. IssuerIDs are immutable and stored in a standardized format to allow cross-verification among peers.
This field guarantees **traceability**, ensuring that any token can be verified back to its origin without requiring a central registry.

6.15.3 RecipientID

The **RecipientID** identifies the target node or user authorized to use the token. This ID must match the identity of the requester in any token-requiring action. A mismatch results in token rejection.
This field ensures that **tokens cannot be traded or hijacked**, as they are cryptographically bound to their intended recipient.

6.15.4 Timestamp & Expiration

Each token includes an issuance timestamp and an expiration window, defining the token's validity period. Once expired, the token is no longer accepted for any operation and is flagged for pruning by participating nodes.
This lifecycle enforcement ensures that **stale tokens cannot be reused** and encourages real-time, verifiable engagement between nodes.

6.15.5 Signature

Every token includes a digital signature generated by the issuing node’s private key. This signature covers all other token fields in canonical order, forming a cryptographic seal of authenticity.
Verification is performed using the public signature key of the issuing node. If the signature is invalid or missing, the token is rejected.

This field provides **tamper-proof guarantees** and enforces the trustless integrity of token-based operations.

The token structure, when combined with peer validation logic and expiration enforcement, forms the backbone of decentralized permission control in the SPHERE framework. Tokens are non-fungible, tightly scoped, and non-transferrable, ensuring that all actions requiring elevated privileges are **explicitly authorized, cryptographically verifiable, and time-constrained**.

6.16 Token Issuance Logic

Tokens are generated and digitally signed only by nodes that have successfully completed the bootstrapping process and are recognized as valid participants within the SPHERE network. A node must possess a valid Contact Record, maintain an operational routing table, and establish cryptographic trust through signed peer interactions before being eligible to issue tokens.

6.16.1 Issuer Eligibility

Only nodes with verified bootstrap status may issue tokens. Bootstrap status is defined as the successful completion of:

- Initial key generation and identity registration,
- Connectivity validation via the DHT (Distributed Hash Table),
- And successful interaction with at least one peer node, confirmed via authenticated handshake.

This requirement is natively built into the issuing process as an action for another node is needed to receive a token, this ensures that only **established and reachable nodes** contribute to permission control, preventing token flooding from unknown or rogue actors.

6.16.2 Conditions for Token Issuance

A token may only be issued after the **recipient node has performed a validated network-supporting action**. Examples of such actions include:

- Responding to a content block request,
- Successfully forwarding or relaying a packet,

- Providing uptime presence across a defined time window,
- Completing a DHT query or route assistance for other nodes,
- Successful completion of a PingPal event.

The issuing node verifies the action through built-in metrics or event callbacks, then signs and issues a new token as a reward or authorization grant.

This condition ensures that token issuance is **performance-based**, tied directly to useful network activity, rather than arbitrary or pre-assigned rights.

6.17 PingPal: A System for Uptime Token Issuance

The PingPal system provides a lightweight, decentralized method for issuing tokens based on sustained uptime. It allows peer nodes to build mutual trust over time by providing consistent availability without centralized monitoring. This mechanism promotes node reliability and rewards long-term participation in the SPHERE network.

6.17.1 Establishing a PingPal Relationship

Each node may select a peer node to act as its **PingPal**. This selection is mutual and voluntary; a node must first request permission to register a PingPal relationship, which the candidate peer may accept or reject.

Once accepted, the PingPal node becomes responsible for storing and validating uptime requests from the initiating node for the duration of the single event lifecycle. After issuance of a token by a **PingPal**, a new **PingPal** is requested.

6.17.2 Initial Ping Request

The node sends a **PingPalRequest**, a signed message indicating the desire to log uptime and initiate the countdown toward token eligibility.
This message includes:

- The sender’s NodeID and timestamp,

- A digital signature,
- And optionally the sender's current uptime streak or heartbeat metadata.

Upon receiving the request, the PingPal:

- Validates the signature,
- Records the sender's ID and timestamp,
- And stores the request in a local list of active uptime monitors.

6.17.3 Uptime Validation and Reward

Exactly 24 hours after the initial ping, the originating node may re-ping the same PingPal, including its original PingPalRequest TokenID and a fresh signature.
If the PingPal:

- Confirms that the same NodeID initiated both requests,
- Verifies the time window (must be >= 24h but within an upper bound),
- And finds no evidence of token misuse or revocation,

Then the PingPal **issues a token to the sender**, using the standard issuance logic described in section 3.2. This token certifies **24-hour continuous uptime**, validated through cryptographic replay-resistant handshakes.

6.17.4 Abuse Prevention and Timing Constraints

If the follow-up ping is missing, submitted too early, or fails signature validation, no token is issued. Nodes attempting to forge timestamps, resend old requests, or reuse expired PingPalRequests may be penalized by having their PingPal status revoked.

Only one uptime token may be earned per PingPal per cycle, and multiple PingPal relationships **cannot be stacked** to bypass this restriction. This maintains fairness, limits token farming, and encourages genuine uptime performance.

While PingPals persist their pal request data across restarts, the requesting peer does not retain PingPal associations after shutdown. As a result, upon restart, the peer must initiate a new PingPal request with a new node, restarting the uptime tracking cycle.

PingPal serves as a distributed timekeeper and trust oracle, enabling nodes to prove they are consistently online over meaningful intervals. This mechanism provides a scalable, low-overhead system for rewarding reliability, complementing event-based token issuance while reinforcing the integrity of the SPHERE network.

6.18 Token Validation and Enforcement

Tokens function as mandatory proof-of-authorization objects for executing privileged actions on the SPHERE network. Before a node may perform sensitive operations such as publishing content, mutating records, or submitting computational tasks, it must present a valid token. The receiving node is responsible for fully verifying the token's structure, authenticity, and constraints prior to accepting the request.

6.18.1 Validation Requirements

When a node receives a sensitive request (e.g., **PUT**, **EDIT**, or **COMPUTE**), it checks for an attached token. If present, the token undergoes the following validation steps:

- **Signature Verification**:
  The token is validated using the **IssuerID** and its associated public signature key. The token’s signature must match the hash of all included fields. Any signature mismatch results in rejection.

- **Recipient Match**:
  The **RecipientID** field must match the ID of the node making the request. This ensures that the token is **non-transferable** and cannot be reused by unauthorized peers.

- **Expiration Check**:
  The current network time is compared against the token’s timestamp and expiration window. Tokens used outside of their valid time range are considered expired and are rejected.

- **TokenID Reuse Check**:
  Nodes maintain a record of previously seen TokenID values. If a token with the same ID has already been used or flagged, it is rejected to prevent replay attacks.

6.18.2 Enforcement Process

Once validation passes, the requested action may proceed. Otherwise:

- The request is rejected with a standardized error indicating invalid or missing authorization.
- The invalid token may be logged and broadcast to peers for anomaly detection (optional).
- Repeated invalid attempts may degrade the requester's reputation or trigger a throttling mechanism.
- **Invalid or malicious behavior results in cumulative reputation loss**, which may lead to eventual node exile from the network, preventing further participation in token-gated operations or peer communication.

6.18.3 Stateless Verification and Performance Considerations

Token validation is **stateless and self-contained**. All fields required for verification are embedded within the token itself, eliminating the need for real-time communication with the issuing node. This design ensures that token-based permission checks are:

- Fast (local public key lookup),
- Secure (cryptographically validated),
- And scalable (no centralized bottlenecks).

Token validation acts as a **gatekeeper** for trust-sensitive operations. It ensures that only authorized nodes may alter network state, submit work, or affect distributed consensus. This model reinforces SPHERE’s goal of enabling decentralized governance through verifiable, cryptographically enforced permission structures.

6.19 Token Lifecycle Management

Tokens are time-bound permission objects designed for one-time use within a defined operational window. To ensure trustless validity and minimize attack vectors such as replay or misuse, tokens follow a strict lifecycle governed by issuance rules, expiration logic, and optional renewal via peer coordination.

6.19.1 Expiration and Time Constraints

Each token includes an embedded timestamp and optional expiration window (see section 3.1.4). Once the expiration threshold is reached, the token is marked as invalid by the receiving node.
Expired tokens are not immediately discarded but are instead added to a **pending removal list**, allowing nodes to:

- Prevent reuse within the decay window,
- Audit expired tokens,
- And assist with network-wide anomaly detection.

This delay window also supports edge cases such as clock drift or network lag while still enforcing a secure expiration policy.

6.19.2 Revocation and Replay Prevention

Tokens are single-use and tracked by their **TokenID** across validating nodes. Any token that has already been accepted, flagged as invalid, or reported by a peer is **considered revoked** and automatically rejected if reused.

Nodes maintain a cache of recently seen or revoked tokens to protect against:

- **Replay attacks**, where an old valid token is reused maliciously,
- **Double submission**, where a token is used for multiple actions,
- Or **conflict injection**, where attackers attempt to poison the validation pipeline.

If a token is marked revoked, all further actions tied to that **TokenID** are denied regardless of signature validity or timestamp.

6.19.3 Extension via PushToken Handshake

To extend a token's validity without full reissuance, SPHERE nodes may engage in a **PushTokenExtend handshake**. This is a two-step ping-pong protocol:

1. The issuing node sends a **PushTokenExtendPing** to the recipient, including the original **TokenID**, expiration timestamp, and a fresh nonce or extension request signature.

2. The recipient replies with a **PushTokenExtendPong**, confirming possession of the original token and revalidating its identity.

Upon successful round-trip:

- The issuing node creates a new token with the same **TokenID** but updated expiration, re-signs it, and delivers it to the recipient.

- The previous version is marked as replaced and no longer valid for use.

This mechanism supports long-lived permission relationships without allowing indefinite validity, preserving both performance and security.

6.19.4 Purging and Cleanup

Tokens marked as **expired, revoked, or replaced** are periodically purged from memory and local tracking caches. This cleanup:

- Prevents memory bloat in long-running nodes,

- Reduces token validation latency,

- And ensures replay protection datasets remain performant.

Nodes may optionally share revocation information in batch via peer-to-peer sync messages to assist with reputation enforcement and blacklist propagation.

Token lifecycle management is designed to preserve trustless permission boundaries while enabling lightweight renewal and secure cleanup. The lifecycle ensures that tokens are **temporally constrained, cryptographically validated, and never blindly trusted beyond their intended window of use**.

6.20 Token Use Cases

Tokens serve as cryptographic gatekeepers for a variety of permissioned actions across the SPHERE ecosystem. Their presentation and validation are mandatory in any context where unauthorized access, tampering, or abuse could compromise the integrity of the network state. The following use cases represent core categories of operations that require token validation prior to execution:

6.20.1 Content Mutation

Tokens are required to perform **PUT** or **EDIT** operations on DHT-hosted content blocks. This includes:

- Publishing new static sites or files to the network,
- Updating existing content hashes,
- Or replacing previously published blocks under the same alias.

Only nodes presenting a valid, unexpired token with matching **RecipientID** may execute mutation requests. This prevents spam, unauthorized edits, and tampering with verified content, while also establishing a **cost-of-action deterrent** through token scarcity.

6.20.2 Identity Mutation

Modifications to a Contact Record—such as updating public keys, display metadata, or authentication tuples—require token-based authorization.

Token validation must pass **before the new contact block is signed and propagated** to the network.

This ensures that identity records remain resistant to unauthorized overwrites or malicious updates, and that all identity changes are traceable to a verified, token-approved node.

6.20.3 Task Submission and Computation Participation

In environments where SPHERE is extended to support decentralized computation or consensus-based task execution (e.g., rendering, data processing, simulation), token presentation may be required before submitting work or participating in a compute quorum.

In these cases:

- Tokens may represent earned compute credits or workload permissions,
- Tokens may also restrict task spam or protect the cluster from overload,
- And validation ensures that only active, reputable nodes consume computation bandwidth.

6.20.4 Alias Registration and Modification

The creation or mutation of human-readable aliases—such as usernames, domain-style references, or symbolic shortcodes—may be gated by token validation.
This includes:

- First-time alias registration,
- Updating alias-to-ID mappings,
- Or reassigning expired or orphaned aliases.

Requiring tokens in this context:

- Prevents namespace squatting and spam,
- Ensures alias creation is linked to real participation,
- And reinforces identity-to-alias trust bindings.

Token use cases extend across all critical write-path operations in the SPHERE framework. Whether publishing, modifying, registering, or computing, token presentation ensures that **each action is deliberate, verifiable, and earned through prior network contribution**—establishing a secure, reputation-aligned operating model without centralized oversight.

6.21 Reputation-Based Earning Model

Token distribution is directly tied to measurable, verifiable, and meaningful network contribution. Rather than relying on traditional economic incentives such as staking or proof-of-work, SPHERE uses a **reputation-aligned reward model**, in which tokens are earned by completing actions that benefit the health, stability, and performance of the decentralized ecosystem.

6.21.1 Task-Based Token Eligibility

Tokens are issued only after a node performs one or more **validated network-supporting actions**, such as:

- Responding to peer pings or relay requests in a timely manner,
- Participating in DHT lookups or content replication,
- Maintaining continuous uptime and connectivity (as measured by the PingPal system),
- Successfully routing packets or assisting with path discovery,
- And in future embodiments, completing distributed computation tasks.

All eligible actions must be verifiable through cryptographic proof (e.g., signed acknowledgments, challenge-responses, or timing logs) before issuance is permitted.

6.21.2 Peer-Issued Validation and Distributed Enforcement

Token issuance is **peer-driven and decentralized**. A node may only receive a token if another node — typically one who directly benefits from the completed task — agrees to sign and issue the token.
Issuers independently validate task completion through event detection, handshake completion, or quorum-based signals. There is **no central authority** managing rewards, and no node may issue tokens to itself.

This enforces **mutual accountability**, ensuring that tokens reflect real-world usefulness as opposed to self-claimed work or synthetic staking.

6.21.3 Integration with Reputation Score

Each node in the SPHERE network maintains an evolving **reputation score**, influenced by both successful actions and observed misconduct. Reputation is increased when:

- Tokens are earned through verified contribution,
- Tasks are completed without delay or failure,
- And peers validate the node's behavior over time.

Conversely, failed handshakes, invalid token submissions, or repeated rejections degrade reputation. Nodes with poor reputation may:

- Be excluded from receiving tokens,
- Be denied permission to mutate data or publish content,
- Or eventually be **exiled from the network** entirely.

Token earning therefore becomes both a reward and a **trust-building mechanism**, reinforcing honest behavior while creating a tangible metric for permission eligibility.

6.21.4 Anti-Gaming Protections

To prevent exploitation of the system, SPHERE includes multiple layers of safeguards that ensure tokens are only issued for authentic, verifiable network contributions. These protections operate at both the cryptographic and topological levels of the protocol.

- **All issuance events are signed and timestamped**, ensuring tokens cannot be duplicated, replayed, or retroactively forged. Each token issuance record can be independently verified by third-party peers.
- **Issuers are penalized for issuing unverifiable or unjustified tokens**. Nodes that issue tokens without corresponding valid activity logs or proof-of-task are flagged and face degradation of their reputation, limiting their future ability to participate in token governance.
- **Cross-validation between peers is encouraged**, especially in multi-node operations (e.g., task quorum, DHT replication). This reduces the chance of fraudulent one-on-one issuance cycles by requiring corroboration from multiple sources.
- **Token farming is rate-limited by time-based constraints**, such as the PingPal 24-hour interval, ensuring that nodes cannot generate excessive tokens in short durations regardless of activity spoofing attempts.
- **Token issuance is geographically and topologically randomized.**
  Nodes have no control over their Kademlia ID, which is cryptographically generated, nor do they choose their peers arbitrarily. Instead, peer interactions are determined by

network proximity, XOR distance, and real-time DHT routing logic. As a result:

  - Nodes cannot preselect friendly peers to collude with,
  - Requests arrive organically based on network topology and routing behavior,
  - And no predictable path exists to bias token flow between specific nodes.

- This decentralized, uncontrollable peer selection ensures that **collusion between nodes is effectively impossible**, even if two malicious actors attempt to coordinate.

SPHERE's anti-gaming protections are built directly into the **fabric of the protocol**. The combination of signature requirements, reputation enforcement, peer randomness, and time-gated issuance ensures that the token economy remains **trustless, merit-based, and resistant to coordinated abuse**—without requiring centralized oversight or arbitration.

The reputation-based earning model transforms SPHERE into a decentralized meritocracy, where access to privileged operations is **earned, not bought**. Tokens reflect proof of helpfulness, and reputation acts as the gatekeeper for authority — reinforcing a resilient, self-regulating ecosystem without central intervention.

6.22 Abuse Prevention and Token Safeguards

SPHERE enforces a multi-layered abuse prevention strategy designed to maintain token integrity, prevent privilege escalation, and eliminate opportunities for replay, impersonation, or fraud. Each safeguard is enforced independently, yet they operate collectively to ensure tokens remain valid **only under tightly controlled conditions**.

6.22.1 Self-Issuance Rejection

Tokens where the **IssuerID** and **RecipientID** fields are identical are automatically invalidated. During validation, nodes reject any token that appears self-issued, even if it passes signature verification.
This prevents nodes from granting themselves unauthorized write access or mutational authority, and ensures that **all tokens must originate from a peer**, maintaining trustless mutual validation.

6.22.2 Issuance Tracking and Non-Reissuance

Each node tracks the tokens it has issued using a **TokenID ledger**, either stored locally or cached temporarily in memory.
Once a token is issued for a specific purpose and recipient, it cannot be reissued or duplicated. If an attempt is made to regenerate a token with the same ID—either by the same issuer or a third party—the system flags it as a replay attempt and **automatically rejects it**.

This mechanism prevents:

- Duplicate use of expired tokens,
- Retrying failed requests using modified timestamps,
- Or token re-creation using partial data from prior sessions.

6.22.3 Expiration Enforcement and Overreach Prevention

Tokens include embedded timestamps and expiration windows (see section 3.1.4). Once expired, a token becomes invalid, regardless of whether it was used.
Attempting to submit expired tokens results in immediate rejection.

Additionally, while this implementation does not use explicit scope fields, **token purpose is inherently defined by the context of the request** (e.g., content mutation, alias update). Nodes enforce usage-context checks by ensuring that tokens are only applied to valid operations associated with their recipient's expected behavior.
This prevents a token from being reused **outside its intended context**, even if the signature and timing are valid.

6.22.4 Replay and Cross-Context Injection Defense

Replay protections are built into both the **TokenID tracking system** and the operation context. Nodes log the use of every token and associate it with the specific request that consumed it. Once a token is accepted for a given operation:

- It cannot be reused, even for an identical operation.
- It cannot be applied to another operation type.
- It cannot be presented to a different node for the same effect.

This prevents:

- Replay attacks across identity domains,
- Duplicate writes or task re-submissions,
- And injection of tokens into unintended code paths or endpoints.

Token abuse is structurally impossible without violating cryptographic constraints, breaching temporal validity, or triggering reputation degradation. These safeguards ensure that every token is **issued once, used once, and verified against strict contextual, temporal, and trust conditions**, preserving the long-term resilience and security of the SPHERE ecosystem.

---

7. Drawings and Diagrams

Figure 1: System Architecture Diagram
Illustrates the core components of the decentralized identity framework, including Contact Records, DHT-based storage, and peer-to-peer authentication pathways.

Figure 2: Login Decryption Flow
Depicts the login sequence using BRACE, showing how a user reconstructs their private key by decrypting obfuscated key segments based on entered credentials.

Figure 3: Contact Record DHT Structure
Outlines the structure of a Contact Record stored in the DHT, including metadata, public keys, encrypted key fragments, and authentication data used for secure identity validation.

Figure 4: Token-Based Publishing Validation
Shows the process of validating a publishing attempt using token-based permissions, including token verification and propagation of a signed content or identity block.

Figure 5: High-level content hosting system architecture:
Illustrates the high-level architecture of the decentralized web framework, including the interaction between the Content DHT, Alias DHT, and Contact DHT layers across participating nodes.

Figure 6: Content publishing and ContentID generation flow:
Shows the process of publishing static content to the DHT, including content hashing, ContentID generation, digital signature application, and storage within the network.

Figure 7: Alias resolution flow from alias to content retrieval:
Depicts the alias resolution workflow, demonstrating how a human-readable alias is converted into a hashed DHT key and used to retrieve the associated content block.

Figure 8: Content mutation flow using BRACE and token validation:
Outlines the content mutation sequence, including user authentication via BRACE, token presentation, and propagation of the updated content block with a new ContentID.

Figure 9: Redundancy and caching propagation between peer nodes:
Shows how content redundancy and caching propagate through the DHT as peer nodes retrieve, validate, and store replicated content blocks for fault tolerance and improved availability

Figure 10: Token Issuance Flow
A diagram showing the process of a node performing a network action, receiving validation from a peer, and being issued a signed token.

Figure 11: Token Structure Breakdown
Visual representation of the token fields, including **TokenID**, **IssuerID**, **RecipientID**, **Timestamp**, **Expiration**, and **Signature**.

Figure 12: PingPal Lifecycle
Timeline view of the PingPal uptime validation sequence: initial request, 24-hour wait, follow-up ping, and token issuance.

Figure 13: PushToken Extend Handshake
Two-way message flow between issuer and recipient to renew a token’s expiration period.

Figure 1: System Architecture Diagram

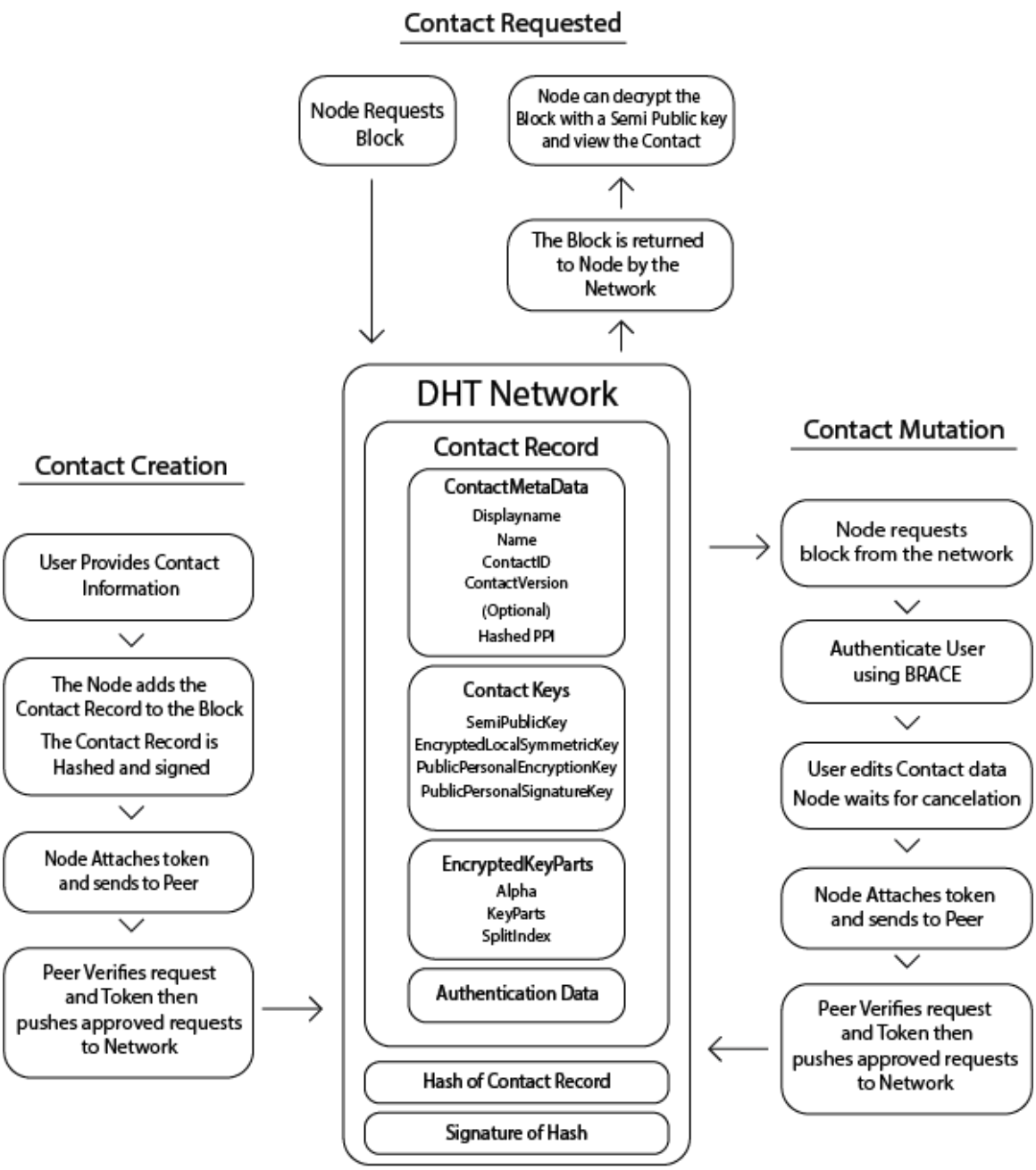

Figure 1: System Architecture Diagram

This diagram illustrates the decentralized identity framework's architecture, including Contact creation, DHT-based storage, mutation via BRACE and token validation, and peer-to-peer block retrieval. Contact Records are stored in the DHT with cryptographic integrity, allowing nodes to authenticate, retrieve, or mutate identity data without centralized servers.

Figure 2: Login Decryption Flow

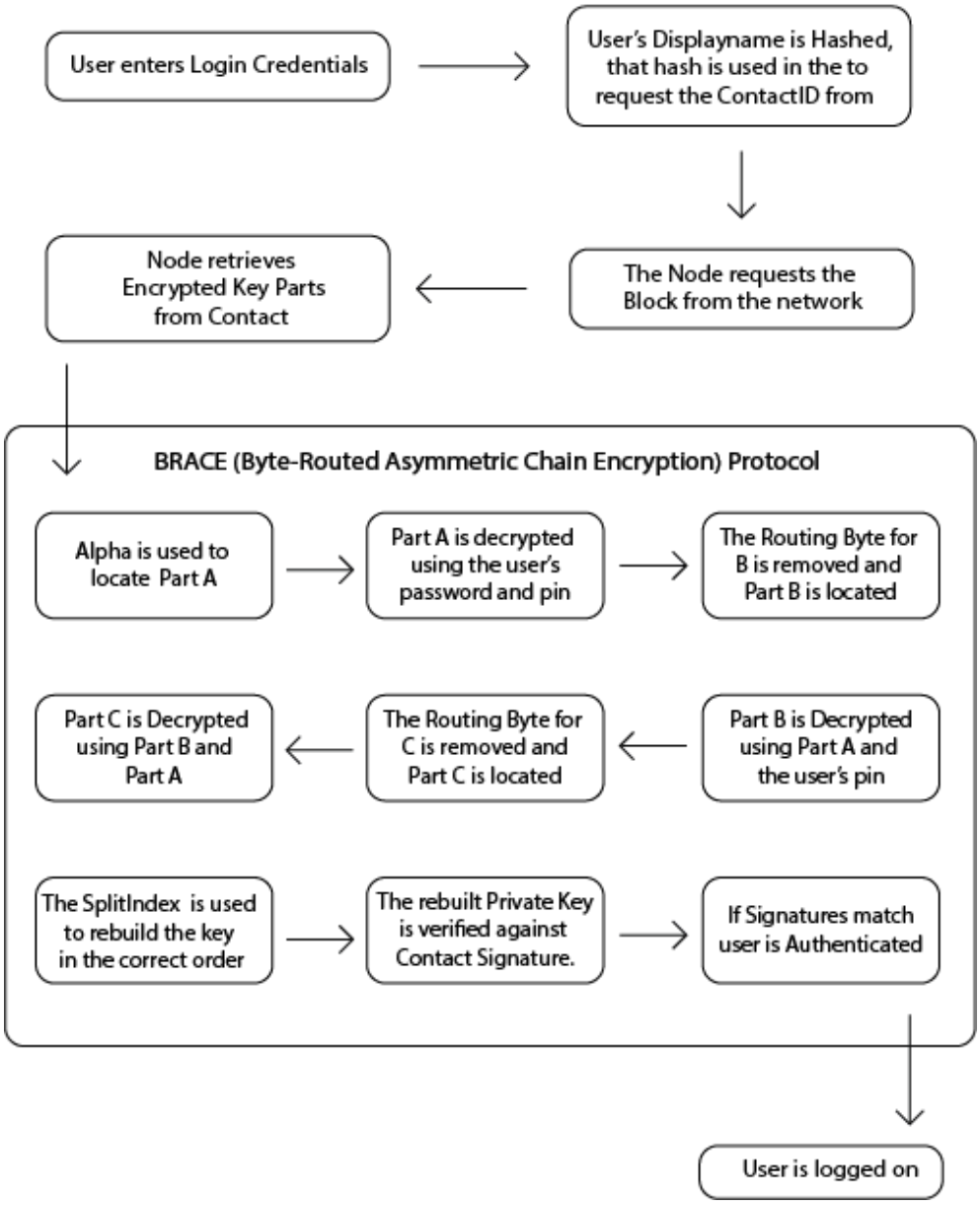

Figure 2: Login Decryption Flow

This diagram illustrates the login process using the BRACE (Byte-Routed Asymmetric Chain Encryption) protocol. After the user submits their credentials, the system retrieves obfuscated key segments from the DHT and decrypts them in a chain-driven sequence. The private key is reconstructed using directional metadata and reassembled via the stored SplitIndex. Authentication succeeds only if the rebuilt key produces a matching signature to the one stored in the Contact Record.

Figure 3: Contact Record DHT Structure

Contact Record Structure	
Contact Meta Data	
DisplayName	A user-facing name or alias.
Name	The user's actual or system-registered name.
ContactID	The unique identifier of the contact (also referred to as the "Block ID").
ContactVersion	A version tag that allows future-proofing and backward compatibility as the contact format evolves.
Keys	
SemiPublicKey	A semi-public identity key used for shared access and contact decryption authorization.
EncryptedLocal SymmetricKey	A symmetric key encrypted using trusted public infrastructure and used to decrypt the contact locally.
PublicPersonal EncryptionKey	Used for encrypting personal messages or data sent to the contact.
PublicPersonal SignatureKey	Used by other peers to validate any data or message the contact signs using their private signature key.
Encrypted Key Parts	
Alpha	A three byte array used to identify the starting point for selection and decryption.
KeyParts	List of all the encrypted parts(True and Decoy).
SplitIndex	Used to track the order the keys need to be placed in for reassembly.

Figure 3: Contact Record DHT Structure

This table describes the internal structure of a Contact Record stored in the DHT. It includes metadata, identity keys, encrypted private key segments, and directional reconstruction parameters necessary for secure login and authenticated mutations.

Figure 4: Token-Based Publishing Validation

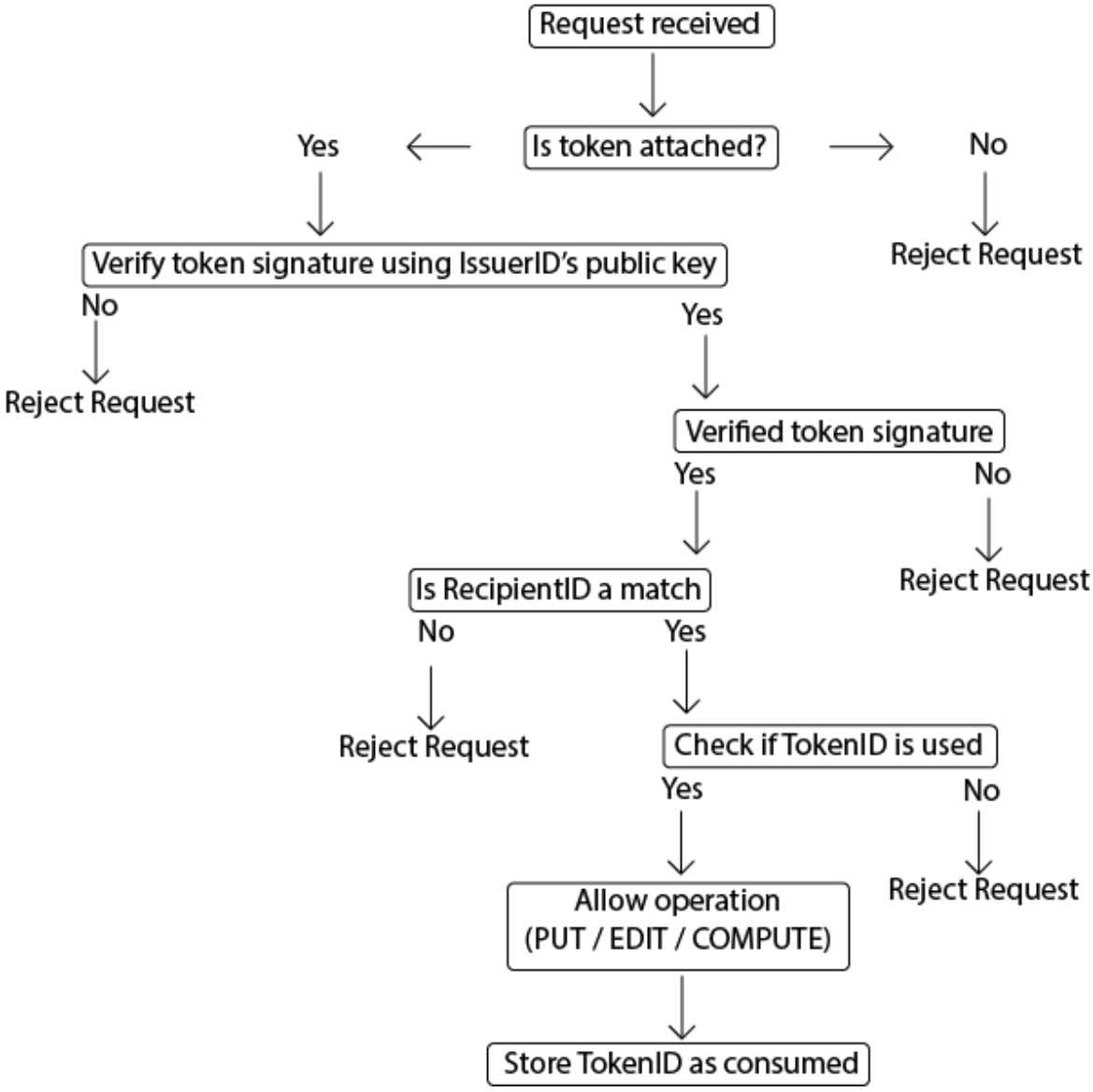

Figure 4: Token-Based Publishing Validation

This flowchart illustrates how token-based permissions are validated prior to allowing sensitive operations such as publishing or mutating a content block. The process verifies that a token is attached to the request, checks the token's digital signature using the issuer's public key, confirms that the token was issued to the requesting identity (RecipientID), and ensures the token has not already been used. Only upon passing all validation steps is the publishing operation authorized, and the token is marked as consumed to prevent reuse.

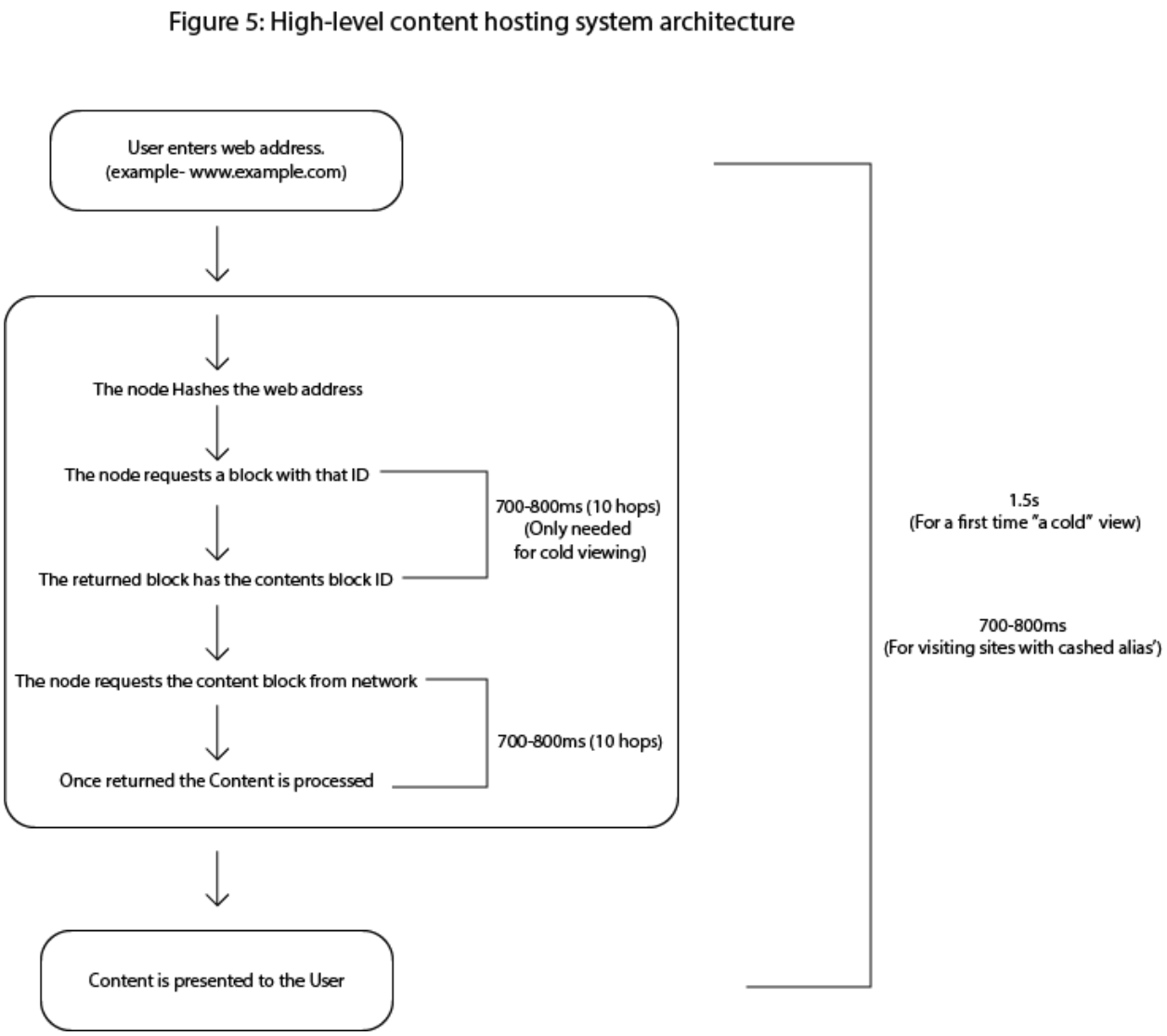

Figure 5: High-level content hosting system architecture

In cold scenarios where the alias is not cached, SPHERE nodes resolve the alias and fetch the associated content block in ~1.5 seconds via two O(log n) DHT lookups. Warm requests (with cached alias) complete in ~700–800ms. Each block retrieval averages 8–10 P2P hops globally, backed by 5x redundancy, enabling consistent access times rivaling traditional centralized CDNs without requiring any centralized infrastructure.

Figure 6: Content Publishing and ContentID Generation Flow

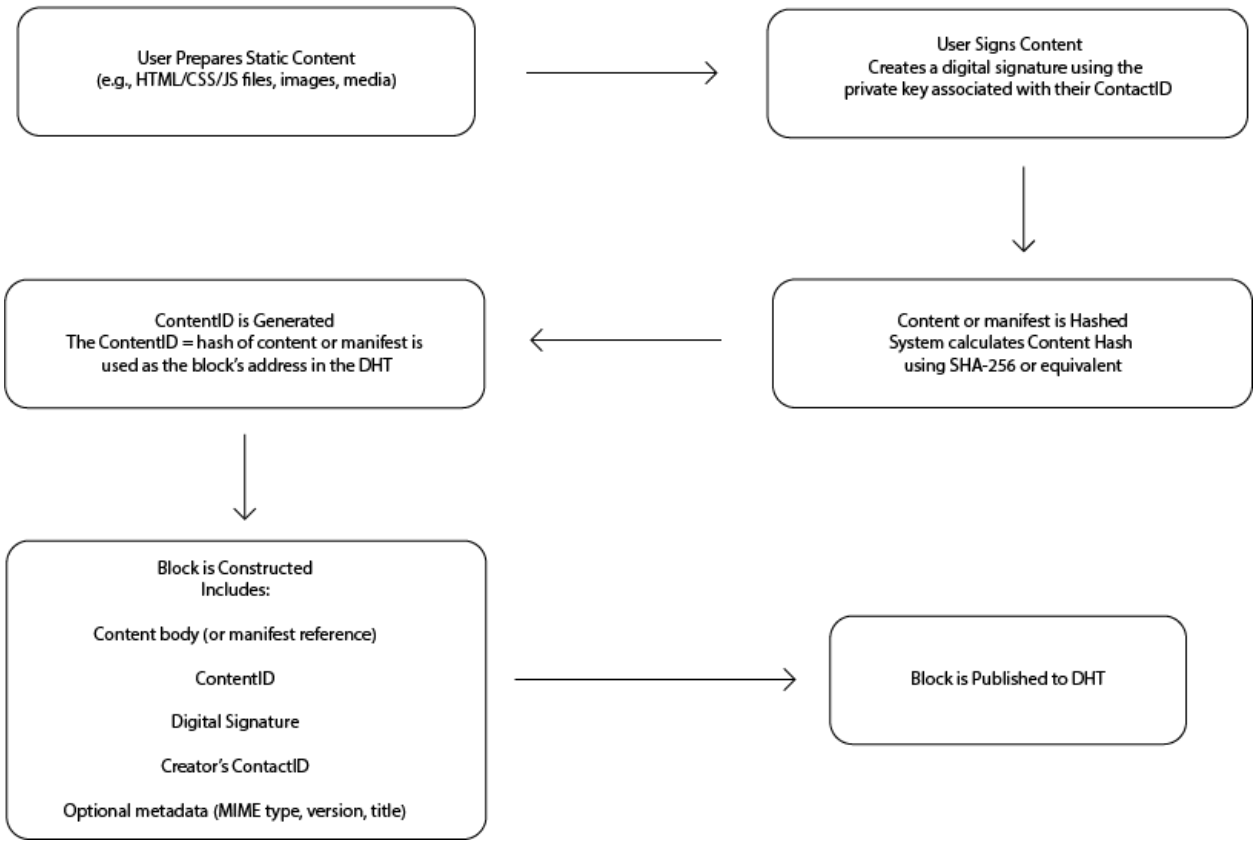

Figure 6: Content Publishing and ContentID Generation Flow

This diagram outlines the process by which user-generated content is signed, hashed, and packaged into a DHT-compatible block. The content is first signed by the author, then hashed to create a deterministic ContentID. The resulting block, which includes the signature and metadata, is published to the network where it becomes globally addressable and tamper-evident.

Figure 7: Alias Resolution Flow from Alias to Content Retrieval

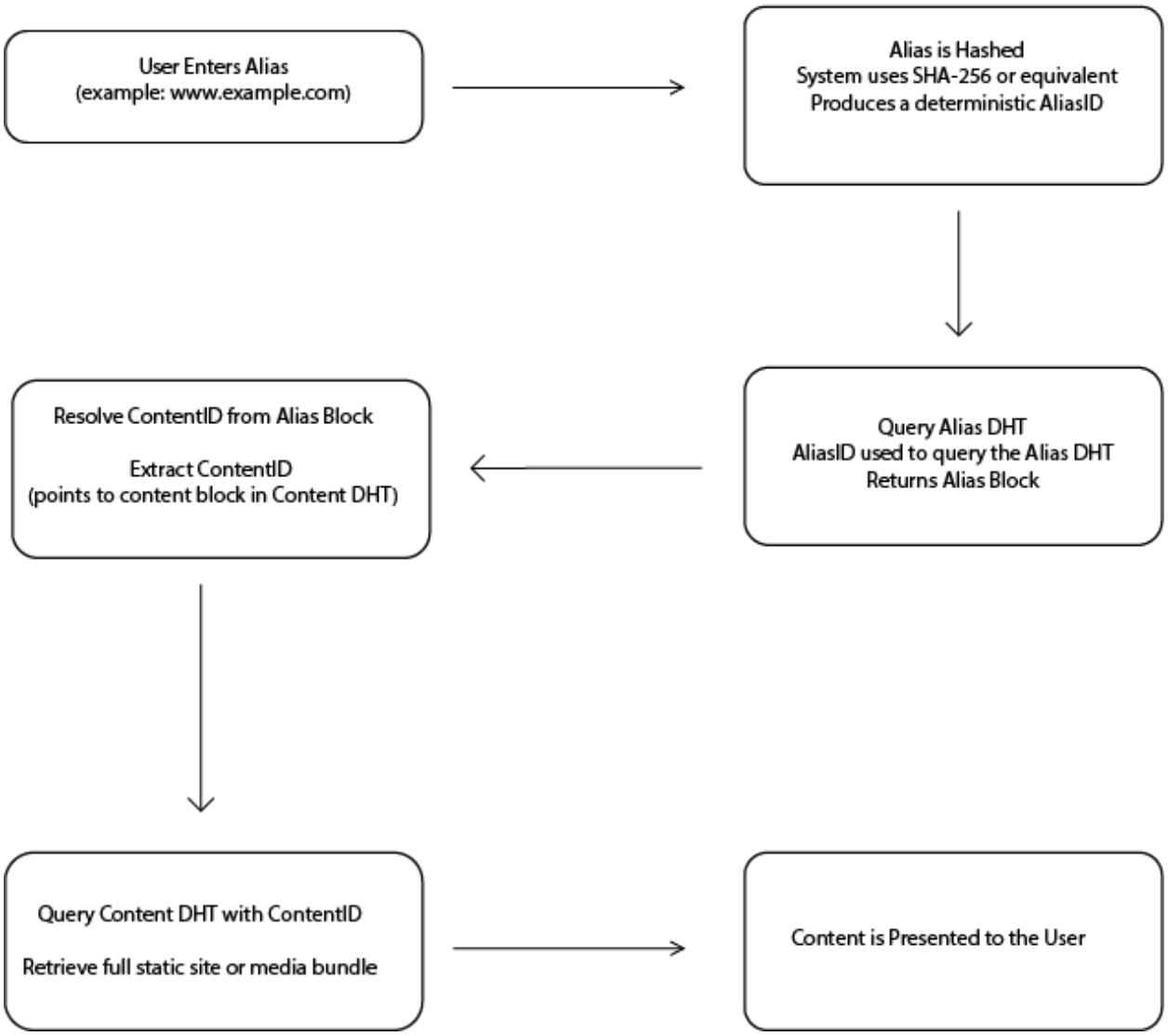

Figure 7: Alias Resolution Flow from Alias to Content Retrieval

This diagram illustrates how a human-readable alias (such as a web address) is resolved to its corresponding content block within the decentralized network. The alias is first hashed into a deterministic AliasID, which is used to query the Alias DHT. The resulting alias block contains the ContentID pointing to the actual hosted data. The system then retrieves the content block from the Content DHT, enabling the user to access the desired site or asset without relying on centralized DNS infrastructure.

Figure 8: Content mutation flow using BRACE and token validation

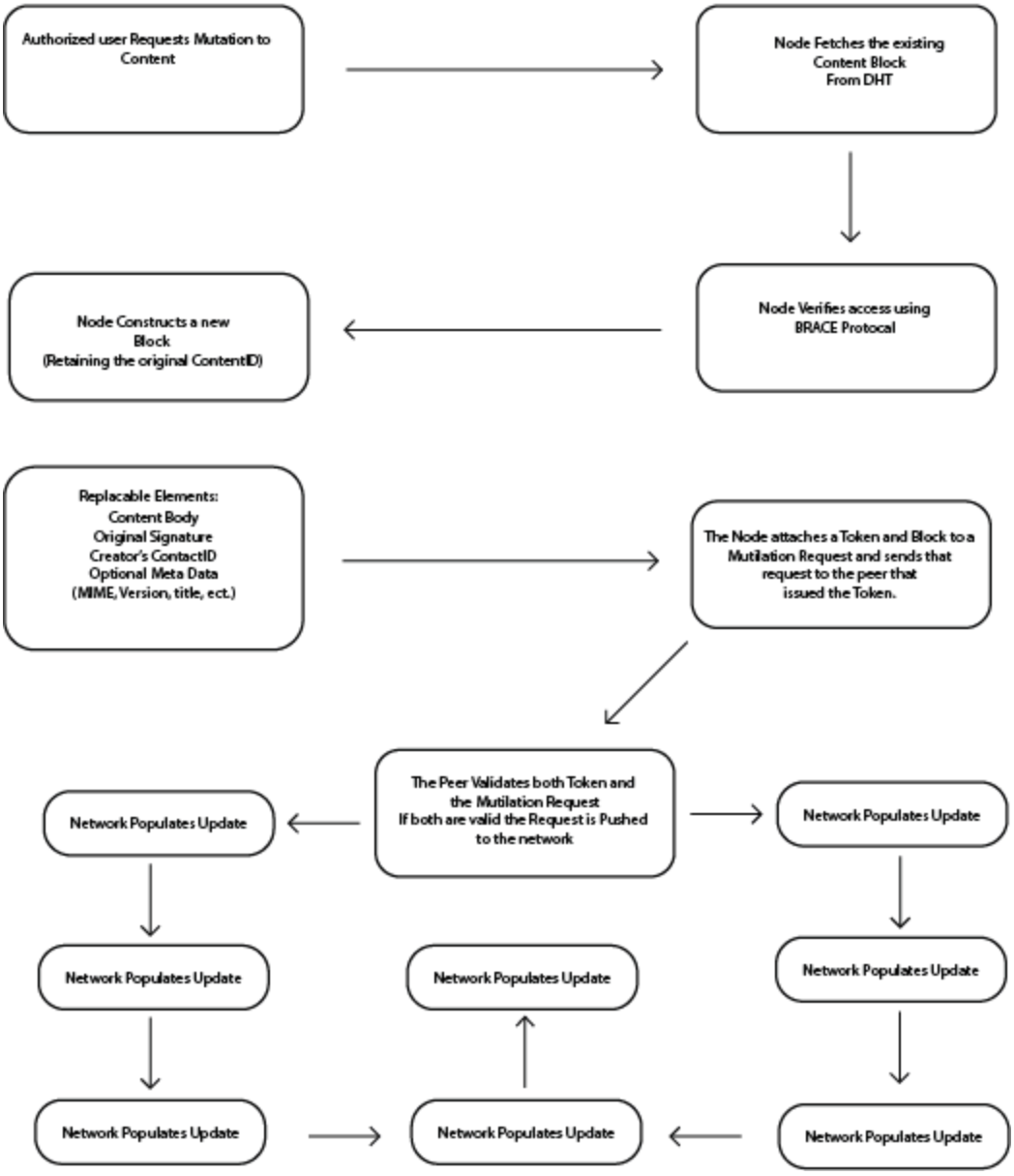

Figure 8: Content Mutation Flow Using BRACE and Token Validation

This diagram illustrates the process through which an authorized user modifies content in the DHT. Upon requesting a mutation, the system fetches the original block, verifies the user's credentials via the BRACE protocol, and constructs a new block retaining the original ContentID. A signed token is attached to the mutation request, which is validated by a peer node. If both the BRACE access and token are verified, the update is accepted and propagated through the network, ensuring only authenticated, token-gated mutations are accepted and distributed.

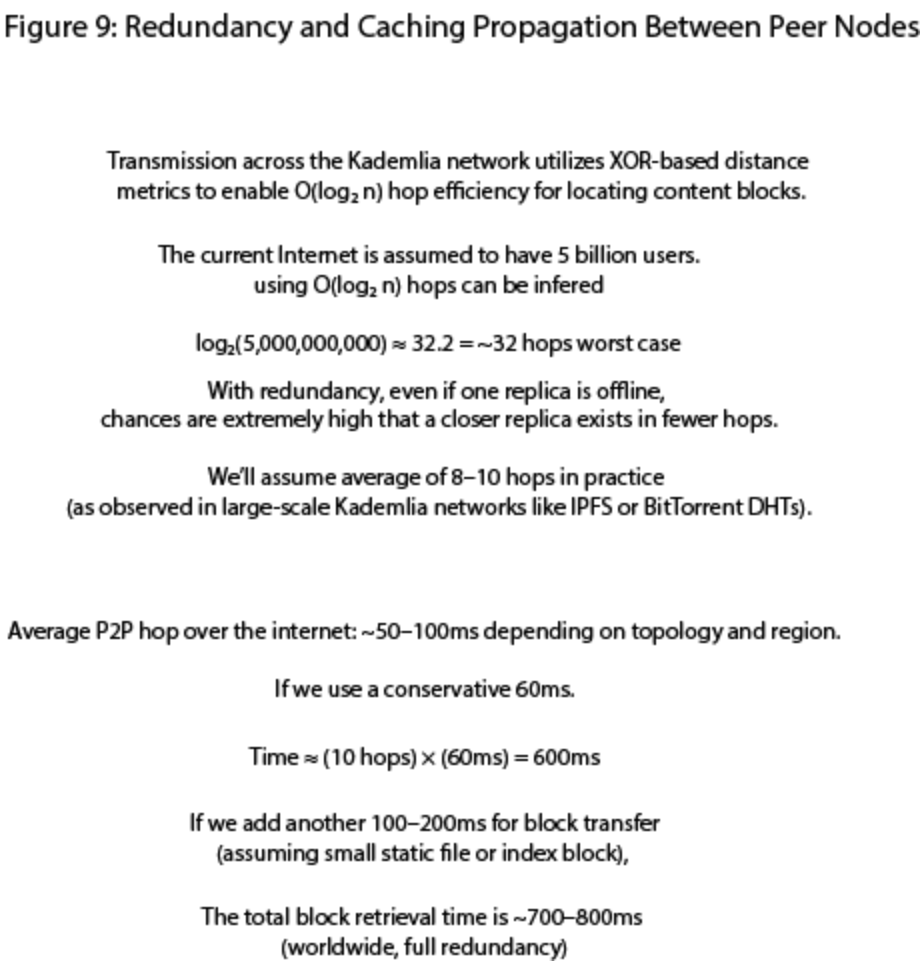

Figure 9: Redundancy and Caching Propagation Between Peer Nodes

This figure illustrates how Kademlia's XOR-based routing enables efficient content discovery with O(log n) hops. Assuming 5 billion nodes, worst-case routing requires ~32 hops, but with 5x redundancy, the average lookup completes in 8–10 hops. Each hop adds ~50–100ms, yielding a typical block retrieval time of 700–800ms. Combined with peer-side caching and deterministic content IDs, this model ensures fast, resilient content delivery at global scale.

Figure 10: Token Issuance Flow

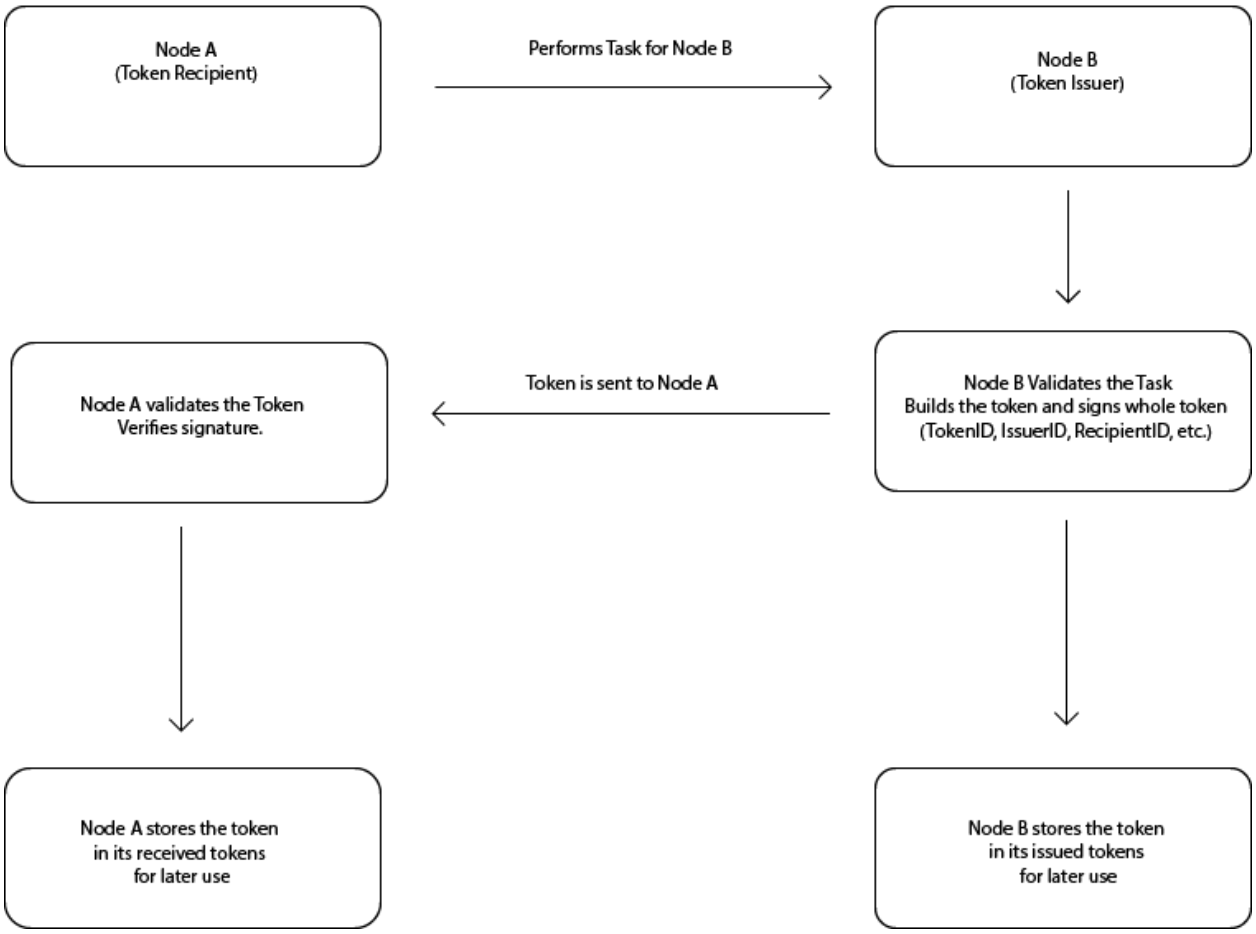

Figure 10: Token Issuance Flow

This diagram illustrates the token issuance process within the SPHERE framework. A node performs a validated task for a peer, which then issues a cryptographically signed token. The recipient verifies and stores the token for future permissioned actions, while the issuer records the issuance for tracking and replay prevention.

Figure 11: Token Structure Breakdown

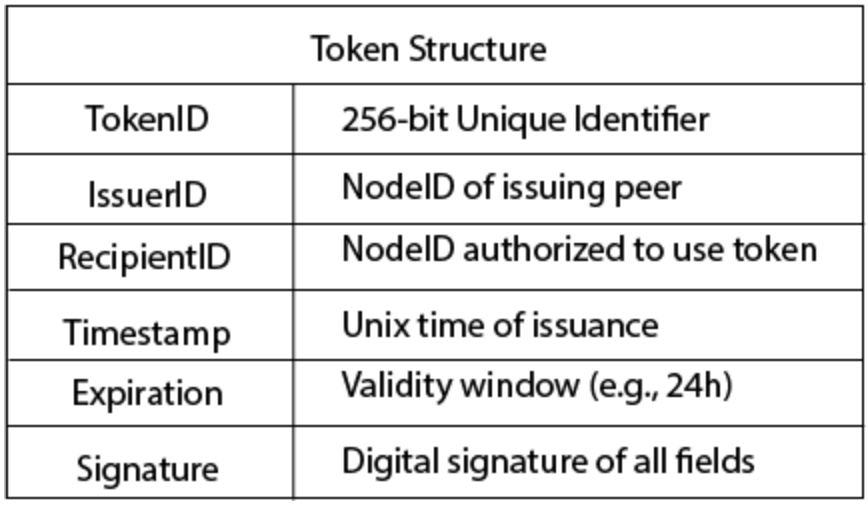

Figure 11: Token Structure Breakdown

This figure displays the structure of a token used in SPHERE, including identity fields, temporal constraints, and a digital signature. Each field is critical for validation, replay protection, and enforcement of decentralized permission control.

Figure 12: PingPal Lifecycle

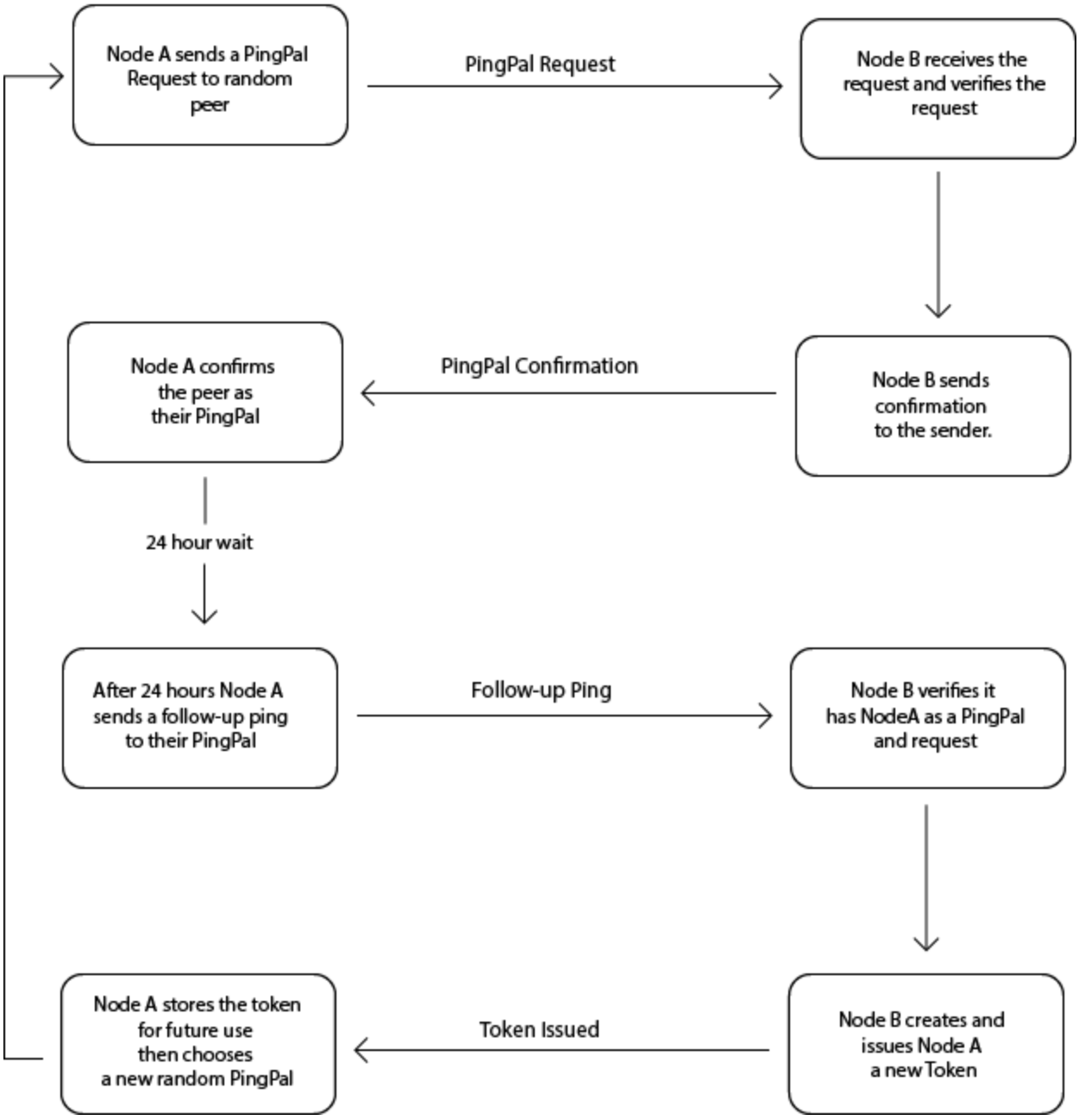

Figure 12: PingPal Lifecycle

This figure outlines the lifecycle of a PingPal relationship, used to validate node uptime. A node requests a PingPal, confirms after peer response, waits 24 hours, and sends a follow-up ping to earn a token. This mechanism ensures non-reusable, time-gated reward issuance tied to verifiable availability.

Figure 13: PushToken Extend Handshake

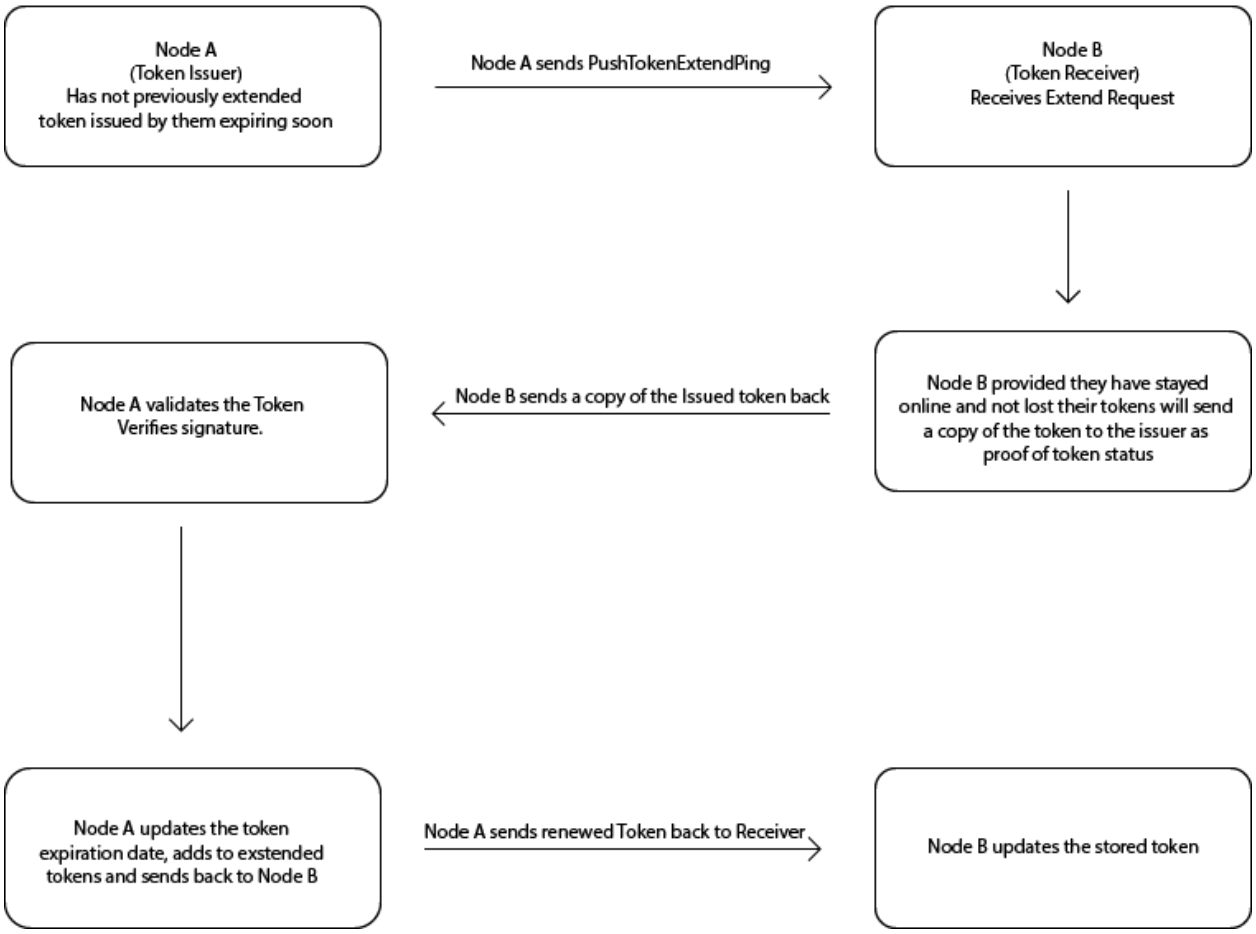

Figure 13: PushToken Extend Handshake

This diagram illustrates the expiration extension handshake for a token between the issuer (Node A) and the recipient (Node B). The process includes verification of token possession, validation of signature, and delivery of a re-signed token with an updated expiration timestamp.

8. Glossary

SPHERE Glossary of Terms

Alias DHT
A parallel distributed hash table used for resolving human-readable names (aliases) into cryptographic identifiers such as ContactIDs or ContentIDs, enabling intuitive lookup in a decentralized system.

AuthenticationData
An array of hash + HMAC tuples used to validate user credentials without transmitting secrets. Includes both valid and decoy entries to obscure inference attempts.

Block
The primary data unit in the SPHERE DHT, used to store encrypted contact records, content, reputation data, or transactions. Each block includes metadata, signatures, and encryption fields.

BlockID
A 256-bit unique identifier for a block, typically derived from cryptographic randomness or deterministic hashing of its content.

Bootstrapping
The initialization process of a node wherein it connects to known peers, populates its routing table, and verifies its identity before participating in the network.

BRACE (Byte-Routed Asymmetric Chain Encryption)
A directional encryption chain for private key segments. Each segment embeds routing bytes for the next segment, hiding true keys among decoys and allowing only the rightful user to reconstruct the private key.

Contact Record
A user’s decentralized identity block, containing public keys, encrypted private key fragments, identity metadata, and authentication data. Serves as the root of all user interactions.

ContactID
The hash-derived address of a Contact Record within the DHT. Used internally for routing and content association.

Content Block
A DHT-stored object representing a static site, file, or media resource. Includes metadata, digital signature, content hash, and linkage to the creator's Contact Record.

Content Hash
A SHA-based digest of the content block or bundle. Mutations produce a new hash, making unauthorized changes easily detectable.

ContentID
A deterministic cryptographic identifier derived from hashing a content object. Used to locate and verify content across the network.

Credential Submission
The process by which users provide their alias, password, and PIN. These are used to decrypt the index and reconstruct key segments.

Decoy Fragments
Fake encrypted private key segments included alongside real fragments to prevent pattern-based inference attacks during BRACE decryption.

DHT (Distributed Hash Table) A peer-to-peer data structure used for storing contact, content, and token blocks. Based on Kademlia's XOR distance metric for efficient retrieval.

EncryptedLocalSymmetricKey
The Local Symmetric Key (used to encrypt a contact or content block) encrypted with the Semi-Public Key. Only accessible to trusted nodes or users with the SPK.

Fingerprint Hash
A hash of the content or contact data used to detect tampering or confirm identity. Often combined with a signature.

Human-Readable Alias
A decentralized username or domain-style handle (e.g., kenny#147) mapped via the Alias DHT to a ContactID or ContentID.

Index Structure
An encrypted object within a contact or content record that dictates the order of key fragment traversal and reassembly. Requires user credentials to decrypt.

IssuerID / RecipientID
Fields in a token that define who created the token and who is authorized to use it. Validation fails if the recipient does not match.

Key Fragments
True and decoy pieces of a private key stored obfuscated within the DHT. Reconstructed using BRACE during authentication..

Local Symmetric Key (LSK)
A secret AES key used to encrypt the block’s data (e.g., contact or content). It is in turn encrypted by the Semi-Public Key.

Mutation Token
A signed, time-bound permission slip allowing the holder to modify a block (e.g., content or contact). Required for PUT or EDIT operations.

Node Types (Full, Power, Mini, Leech)
Categories of node roles:

- **Full**: Stores full DHT and manages critical consensus.
- **Power**: Stores large shards or entire small networks.
- **Mini**: Stores typical shards for lightweight clients.
- **Leech**: Participates without storing data (read-only).

PingPal
A peer node selected to validate your uptime by recording a ping and confirming its repetition after 24 hours to issue a token.

PrivatePersonalEncryptionKey
Used for decrypting messages encrypted with the user's public key. Protected by BRACE and never stored in plain form.

PrivatePersonalSignatureKey
Used to sign contact and content blocks. Proof of authorship and the foundation for all identity validation.

Proof-of-Work (Eco-Mining)
A green alternative to traditional mining. Nodes earn tokens by routing, storing, or verifying data rather than wasting computational energy.

PublicPersonalEncryptionKey
Used by other nodes to encrypt messages to the user. The user decrypts these with their private counterpart.

PublicPersonalSignatureKey
Used by others to verify data signed by the user. Linked to the user's Contact Record and included in block headers.

PushToken
A signed object given by one peer to another as proof of work or permission. Required for sensitive operations like PUT or EDIT.

PushToken Extend Handshake
A two-way protocol in which a token's validity is renewed through an authenticated ping-pong exchange between issuer and recipient.

Redundancy and Caching
Mechanisms for fault-tolerance where blocks are stored by multiple peers to ensure data survival and reduce fetch times.

Replay Attack
A malicious reuse of a valid token or message. Prevented in SPHERE by caching TokenIDs and disallowing duplicate usage.

Routing Table
A list of peers held by a node, sorted into buckets by XOR distance from the node's ID. Enables fast lookup and DHT traversal.

Routing Bytes
Three leading bytes in a BRACE-encrypted fragment used to identify the next fragment in the decryption chain.

Shard
A segmented portion of the DHT assigned to a node. Sharding allows scalability and distributed load handling.

Signature Validation
The process of verifying that data (blocks or tokens) were signed by the correct private key using the attached public key.

SplitIndex
A numeric indicator of the original shuffle order for key segments (Alpha, Beta, Delta). Required to reassemble the private key.

Subnet Overlay
A scoped mini-network within SPHERE defined by a unique namespace or bootstrap key. Useful for private groups or organizations.

Tamper Resistance
The cryptographic immutability of blocks. All mutations create new hashes, and all blocks are verified for integrity and signature.

Token
A cryptographically signed structure that grants access or permission to perform a specific action on the network.

Token Expiration
The enforced time after which a token is no longer valid, preventing infinite or delayed use.

TokenID
The unique identifier of a token, used to track usage and prevent replay attacks.

Token Pruning
Automatic removal of expired or used tokens from memory, improving efficiency and enforcing one-time use.

Uptime Token
A PushToken granted to a node after it maintains presence for 24 hours and completes a PingPal cycle.

---

9. Patent References

This work summarizes the architecture, mechanisms, and methods described across the following provisional patent filings:

Patent Filings (Pending):

- U.S. Provisional Application No. 63/807619 — **System and MethoDecentralized Identity Authentication and Contact Storage Using Encrypted Key Segments**

- U.S. Provisional Application No. 63/807657 — **System and Method for Decentralized Content Hosting and Address Resolution via Distributed Hash Tables**

- U.S. Provisional Application No. 63/807621 — **System and Method for Token-Gated Permission Control and Resource Validation in a Decentralized Network**

Additional claims covering authentication, content mutation, peer-issued governance, and token lifecycle control are protected under U.S. provisional applications filed in 2025.

---

10. Executive Overview: What is SPHERE?

SPHERE is a decentralized protocol for identity, content hosting, and access control—built to eliminate reliance on central servers, password databases, and gatekeeping platforms.

Instead of using usernames and passwords stored by corporations, users authenticate using encrypted, obfuscated fragments of their private keys stored in a distributed hash table (DHT). Only the rightful owner can reassemble their key to log in, update their profile, or publish content.

All content is stored in peer-to-peer blocks, addressable via cryptographic hashes and human-friendly aliases. SPHERE provides full censorship resistance and tamper detection. Updates require **peer-issued tokens**, which are earned through helpful behavior—like staying online, routing data, or validating peers.

There are no miners, no master keys, and no server logs. Everything is cryptographically verifiable. The result is a resilient, scalable network where users—not companies—control their identities, data, and reputation.

Whether you're publishing a website, managing your digital identity, or participating in decentralized computation, SPHERE enables secure, verifiable participation—without the need for centralized infrastructure.

---

11. Copyright Notice